RESEARCH AND DEVELOPMENT TECHNICAL REPORT
CECOM-TR-91-Q502-F /(SED No. C13-091MB-0001-00)

REPRESENTING MULTIPLE REQUIREMENTS TECHNIQUES

Kathleen A. Jordan
Alan M. Davis

Telos Corporation
55 North Gilbert St
Shrewsbury, NJ 07702

October 1991

Final Report for Period October 1990 - October 1991

DISTRIBUTION STATEMENT

Approved for public release;
distribution is unlimited.

CECOM RESEARCH, DEVELOPMENT, AND ENGINEERING CENTER
SOFTWARE ENGINEERING DIRECTORATE

US ARMY COMMUNICATIONS-ELECTRONICS COMMAND
FORT MONMOUTH, NEW JERSEY 07703-5000

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

The citation of trade names and names of manufacturers in this report is not to be construed as official Government indorsement or approval of commercial products or services referenced herein.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704 - 0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	October 1991	Final Report: Oct 90 to Oct 91

4. TITLE AND SUBTITLE
REPRESENTING MULTIPLE REQUIREMENTS TECHNIQUES

5. FUNDING NUMBERS
C: DAABO7-91-D-Q502
PE: 63783A/A094
PR: Six
TA: 085-1-02006-4261

6. AUTHOR(S)
Kathleen A. Jordan* and Dr. Alan M. Davis*
(*George Mason University)

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESSES
Telos Corporation
55 North Gilbert Street
Shrewsbury, NJ 07702

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
US Army Communications - Electronics Command (CECOM)
Research, Development, and Engineering Center
Software Engineering Directorate
ATTN AMSEL-RD-SE-R-ESD-ST (Harlan Black)
Fort Monmouth NJ 07703-5207

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
CECOM-TR-91-Q502-F
C13-091MB-0001-00
- (SED)

11. SUPPLEMENTARY NOTES
Kathleen A. Jordan and Dr. Alan M. Davis were with the Center for Software Systems Engineering, George Mason University, Fairfax VA.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

Currently, there exist dozens of notations and techniques for the specification of software-level and system-level requirements. While there are unique aspects to every requirements notation and technique that make it appropriate for a particular use, there are also areas of commonality between them. To address the proliferation, a requirements model has been developed that singularly expresses the elements in multiple requirements notations and techniques. The objective of this model of models, or "metamodel," is to capture the semantics of all requirements notations and techniques.

14. SUBJECT TERMS
Software Requirements; System Requirements; Software Methodology; Requirements Engineering; Rapid Prototyping

15. NUMBER OF PAGES 59

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclass	Unclass	Unclass	Unlimited

Table Of Contents

List of Figures

List of Tables

SECTION 1

1. Introduction

1.1 Purpose of Document

This document defines and describes a set of requirements engineering primitives that singularly expresses the elements of the many requirements engineering notations in use today. The objective of this metamodel is to capture the semantics of all requirements analysis and specification techniques.

1.2 Background

This metamodel is intended to support the requirements engineering process. Requirements engineering is defined as an activity to determine and specify the external functionality and attributes for a system. This activity consists of two parts: (1) analysis of the existing problem, and (2) specification of the requirements of the intended system solution [DAV90a]. Although requirements engineering is usually associated with the initial stages of system and software development, it is also an ongoing activity throughout the development life cycle. The major product of this activity is a requirements specification that should communicate to both the system users and the system developers the behavior and attributes of the desired system. Since the quality of the eventual system rests substantially upon the quality of the requirements specification, this specification should be understandable to the nontechnical system user and unambiguous to the system developer. It should also be internally consistent.

1.3 Rationale

The motivation for developing a comprehensive requirements metamodel is that there currently exists a plethora of techniques for the analysis and specification of system and software requirements. These techniques purport to do basically the same thing: describe the externally visible attributes and functionality of a software system. Each technique utilizes one or more requirements notations. These notations range from state-based (e.g., R-nets, stimulus-response sequences, statecharts) to process-based (e.g., data flow diagrams, PAISLey) to entity-based (e.g., entity-relationship-attribute, object-oriented analysis diagrams). These notations, however, are limited in that each presents only a partial set of requirements. Thus, in specifying requirements, the use of more than one notation is often necessary since no single notation can capture all aspects of requirements. Meyers [MEY91] notes that "it is often desirable to simultaneously view a system in more than one way; this is especially true when many people are working on the same project. As a result, there is a crucial need, not only for programming environments that facilitate the creation of multiple views of software systems, but for integration mechanisms that let those views work together effectively." A mechanism that provides multiple views could serve as a basis for integrating the multiple notations that are necessary for a complete specification of requirements.

One question is whether there is sufficient overlap between the notations to enable the integrating mechanism to provide useful inconsistency reports. The answer to this is "yes," as we shall see clearly in Sections 3 and 4. For instance, both data flow diagrams (DFD) and object

oriented analysis (OOA) diagrams can represent process-like concepts, i.e., transforms in DFDs and services in OOA. This report will describe how these two concepts are integrated.

A second question is whether different readers of the requirements specification require different notations. The answer to this is also "yes." The system user looks for requirements in the language of the application domain, whereas the system designer looks for requirements in terms of the software implementation domain. Although these views can be very different, a requirements specification should reflect these alternative perspectives, being understandable to a system user and unambiguous to a system designer. Given these distinct views, it would be helpful if we had a way of relating or "connecting" them. A recent workshop [BLA89] identified the need to support these multiple views.

1.4 Related Research

The need for multiple views has been recognized within the software community. Research has been conducted in the development of environments that allow an analyst several choices (i.e., multiple views) of specification notations.

1.4.1 Requirements Language Processor

One of the earliest efforts in requirements environments was the Requirements Language Processor (RLP) at GTE Laboratories in Waltham, Massachusetts. This system was part of a larger system, Requirements Processing System (RPS) [DAV79, DAV80]. To capture and represent requirements, RLP took a language-based approach. It was "multi-viewed" in that it processed requirements expressed in the particular language of the application domain; hence, it was able to process multiple languages. The resulting specification, which took the form of structured English, was then parsed into a corresponding finite state machine representation and stored in a database. This database then provided the input to the subsequent phases of the development life cycle. The multiple views of RLP were from the perspective of the application, not the varied views from different notations.

1.4.2 Knowledge Based Software Assistant

A second and very prominent effort has been the Knowledge Based Software Assistant (KBSA) project at Rome Air Development Center, Griffis Air Force Base, New York. The objective of the KBSA environment is to provide automated, software development support throughout the entire life cycle. Part of the KBSA is a "Requirements Assistant" (RA) that supports the requirements analyst in the capture of requirements. The KBSA/RA concept consists of four "presentation modes" [ELE89]: intelligent note pad, graphical presentations, calculator-spreadsheet, and requirements document. The graphical presentations mode supports the use of data flow diagrams, functional decomposition diagrams, state transition diagrams, and internal interface diagrams.

One of the KBSA/RA objectives was the support of an individual preference of methodology, where using a single knowledge-based database, a specification entered in one format would be "reflected" in the other available formats. According to Abbott [ABB89], the KBSA/RA prototype requires further improvement, particularly in processing inputs through the "intelligent note pad" option. He did not address, however, the KBSA/RA progress in

transforming one graphical view to another, e.g., data flow diagrams to state transition diagrams. A recent workshop [BLA89] identified the need to support these multiple views.

1.4.3 Requirements Engineering Environment

The Requirements Engineering Environment (REE) is an effort by the Air Force to "improve the quality and accuracy of the requirements identification process for Air Force C3I system/software" [ISS90]. The REE is a combination of several tools which include rapid prototyping tools, database, graphical editors, and a modeling tool to construct four different types of performance models. These performance models are operator, functional, communication, and processor. The operator model provides the capability to assess timing and performance issues related to host hardware and user-system interaction. The processor model simulates the target architecture, assessing memory, I/O, and terminal configurations. The communications model provides analysis of network loads resulting from message traffic in a distributed communication system. The functional model is used for modeling the events, procedures, and functions and their effect upon system resources.

1.4.4 FSSB Software Standards and Procedures

The need for multiple views has also been recognized by the Computer Sciences Corporation in their development of Flight Software Systems Branch (FSSB) Software Standards and Procedures (FSSP) [CSC90]. This standard outlines an Object-Oriented Requirements Analysis (OORA) methodology that uses four different models for the analysis and specification of requirements. The four models are (1) the "information model," (2) the "state model," (3) the "process model," and (4) the "object communication model." These four models constitute the OORA methodology, and as noted in [CSC90], requirements are defined using the models in the order given above.

In the information model, the user defines objects, attributes, classes, and subclasses. This initial description is the basis for the subsequent models. Next, using the state model, the states of the dynamic behavior of each object are defined. The interactions of objects are captured as events that compose an event list. The process model then uses DFDs to define the processes within each state of the state model. The standard makes the point that DFDs are not used here as in traditional structured analysis methods. That is, DFDs do not serve as the basis for system decomposition; instead, the information model provides the system decomposition using objects. Lastly, the object communication model combines and summarizes the interactions between the objects based upon the events defined in the state model.

An interesting aspect of this work is the choice of models used within the standard. The standard included an object-oriented technique, a process-based technique, and a state-based technique. Although this was done in the context of an object model (i.e., information model), it nevertheless shows that these particular views are sufficiently distinct (and important) to be included in this methodology.

Although this methodology provides a multi-faceted view of requirements, there are two major differences between this and the metamodel. First, the FSSB methodology leads to an object-oriented model that details within itself the processes and states that can occur. The metamodel is not an object-oriented model; it can (or should) accommodate requirements

organized using any approach. Secondly, the FSSB methodology is not intended as a way of deriving one view from another. One of the objectives of the metamodel is to have the ability to derive, at least partially, one requirements view from another.

1.4.5 New TRADOC Pamphlet

A recently drafted U.S. Army pamphlet [ARM91] has expressed the position that requirements need not be organized in the traditionally used functional decomposition. As alternatives, it suggests object-oriented, stimulus-oriented, response-oriented, feature-oriented, and state-oriented. Although it stops short of recommending multiple techniques on any one system, it does make it clear that no one notation is sufficient for all system developments.

1.4.6 Rumbaugh

James Rumbaugh, et al., [RUM91] have suggested a methodology that uses three views: object, finite state machines, and data flow diagrams. However, this method has no underlying integration mechanism or common representation, and thus lacks the ability to expand to additional views.

1.4.7 Harel

David Harel [HAR87] has suggested a methodology that uses three views: statechart, data flow diagrams, and structure charts. However, this method also has no underlying integration mechanism or common representation, and thus lacks the ability to expand to additional views.

1.5 Integrating Multiple Views

1.5.1 Lucite Box

One way of understanding the different perspectives presented by the diversity of requirements notations is to visualize a box with clear sides. Each side provides a different notation or view of requirements (see Figure 1-1). While the complete set of requirements is inside the box, one can see only the requirements projected by a particular view. Thus, one side of the Lucite Box might be a DFD view while another side might be an OOA view while yet another side provides an FSM view of the requirements. To see the complete set of requirements we need multiple views. This concept of multiple views of requirements is called the *Lucite Box* and was introduced by Davis and Jordan in [DAV91]. Although requirements can be viewed from the outside only, inside the box there exists a model of requirements that integrates all possible external views. This internal model is the requirements "metamodel," and it provides the connections between the distinct, external views of the available requirements notations.

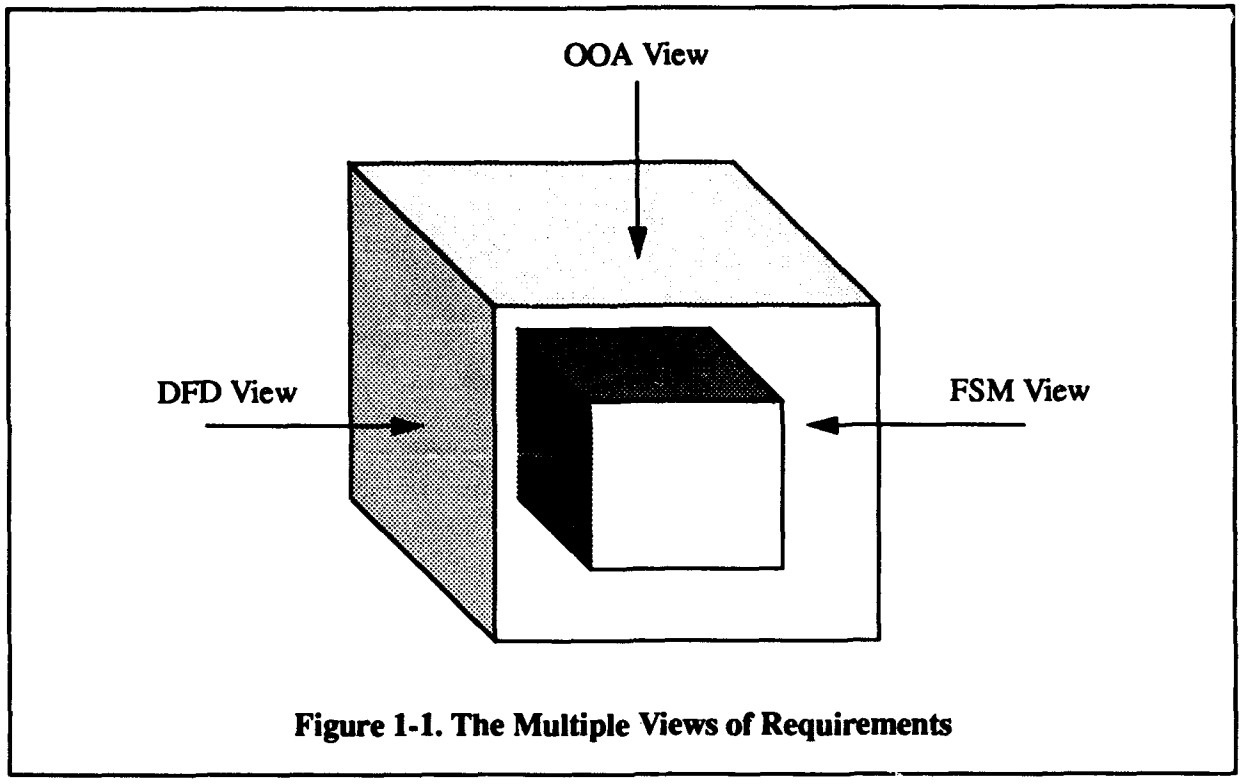


Figure 1-1. The Multiple Views of Requirements

1.5.2 Mechanism for Multiple Views

The Lucite Box provides a conceptual perspective in understanding multiple views, primarily that these views (sets of requirements) are not completely disjoint and do have points of intersection. The Lucite Box works on the premise that, in theory, a single representation of requirements is possible. This single representation would constitute an elemental or canonical view. Although it would be complete, it would not be suitable for requirements writers to use directly due to its complexity. The metamodel is not intended to be used by the requirements writer but only as an internal representation to aid in determining problems in a specification written using more familiar notations. Meyers [MEY91] notes the "allure of a canonical representation - in some sense the Holy Grail of environment integration" to provide the basis for environment, tool, and hence view integration.

1.5.3 Benefits of an Integrated Model

As noted originally in [JOR91], a metamodel representation of requirements could substantially enhance the development of software requirements in a number of different ways. First, a requirements writer would be able to switch from one notation to another without significant loss of work. For example, if requirements were specified in a DFD notation, then switching to OOA would not result in a complete loss of the DFD specification. This factor could make the requirements specification process more adaptable to change. Second, the requirements writer would be able to use more than one requirements notation and maintain consistency between two different representations of the same requirements. Two requirements writers would be able to select the most appropriate requirements notation for the particular system aspect being

specified. Given such a specification written using multiple notations, a requirements writer would be able to compare different views of the same specification. Lastly, a requirements writer would be able to derive one view from another and would not have to be knowledgeable in all the techniques that exist. A specification in statecharts or Petri nets, for example, could be "translated" into a corresponding finite state machine.

SECTION 2

2. Metamodel Definition

2.1 Metamodel Development

This section discusses the development of the metamodel, the approach taken, and the resulting derivation of the metamodel primitives. The metamodel was defined in two phases. The first was intended to be a strawman and was based on just three requirements notations. The second is a complete metamodel.

2.1.1 Phase 1 - Initial Approach to Metamodel Development

Since the metamodel must be comprehensive enough to represent all possible requirements notations, the approach in this research was to use three diverse notations to define the initial metamodel. The criteria used for the selection of the three initial requirements techniques were that they were (1) in common use by the requirements community, and (2) substantially diverse in the concepts represented. In this way the initial definition of the metamodel would be reasonably comprehensive. The three techniques chosen were DeMarco Data Flow Diagrams (DFD) [DEM79], Coad's Object-Oriented Analysis (OOA) [COA91], and finite state machines (FSM). Collectively, they capture an emphasis on object (OOA), process (DFD), and state (FSM).

The methodology in constructing the metamodel consisted of the following steps:

1. comparison of the three techniques for common and unique characteristics;
2. definition of the metamodel, its elements and relationships including the selection of a notation;
3. establishment of the mappings from the requirements notations to the metamodel and vice versa.

2.1.2 Phase 2 - Additional Techniques

The second major phase of metamodel development was the incorporation of six additional requirements notations. These notations were Harel's statecharts [HAR87], decision tables, decision trees, Petri nets, and the Ward and Hatley variants of Structured Analysis [HAT87, WAR85]. Statecharts have extensions for including conditions under which state transitions can occur and also provide concurrent states. The incorporation of decision tables and trees verified that the decision-making logic inherent in the metamodel worked. The addition of Petri nets strengthened the specification of behavioral aspects including timing and synchrony. Lastly, by including other variants of Structured Analysis, aspects such as discrete and continuous data flow, activation signals, and control processes were added, and we further expanded on the relationships between finite state machines and DFDs.

2.1.3 Deriving the Metamodel Primitives

Given these requirements notations, what elements should constitute the metamodel? One approach is to create a metamodel element for every separate element found in each requirements notation. A major problem with this approach is that as other notations and models are incorporated into the metamodel, the metamodel will become extremely large. Another problem is that having each piece of each notation represented separately is not helpful in our desire to reconcile multiple views. The multiple views would be captured by such a model, but each view would be disjoint from the others. The desire for finding "intersecting points" between the various requirements notations is one of the motivations behind the metamodel.

Therefore, given these requirements notations, it was necessary to compare the characteristics of each one. What concepts were shared between them and what concepts were unique? For those concepts that appeared similar, were their representations and/or their semantics consistent in all cases? By identifying unique concepts and eliminating redundant ones, a set of primitive conceptual elements was derived from the nine notations. It was also necessary to note the types of relationships, either implicit or explicit, that were allowed within each model. For instance, could entities be grouped together? Could they communicate with each other?

2.2 Metamodel Notation

As noted above, the elements of metamodel were derived by comparing and contrasting the nine requirements notations. To describe and define the metamodel, a graph notation was selected of labeled typed nodes interconnected by typed arcs. The labels enable us to identify uniquely a particular element (e.g., the "customer" entity). The types enable us to recognize that a particular element is of a predetermined category with predefined characteristics and behaviors.

In the graph notation, the basic elements of the metamodel are represented as nodes, and the relationships between these elements are represented as arcs. In addition, if a relationship has precedence or some implied direction, then this relationship is shown with a directed arc, from the superior to the inferior.

Finally, there are a few nodes that are created within the metamodel with very specific meanings (i.e., the same way that BEGIN has special meaning in PASCAL). These special nodes are: attribute nodes with names START (see Section 3.5.2), DISCRETE (see Section 2.3.8), CONTINUOUS (see Section 2.3.8), DATA (see Section 2.3.8), or CONTROL (see Section 2.3.8), and relation nodes with Boolean names (see Section 2.3.4).

The following section describes the elements (nodes) and relationships (arcs) of the metamodel. The types of metamodel nodes include entity, process, state, attribute, message, constant, relation, and state transition. The types of metamodel arcs include association, ownership, composition, instantiation, communication, stimulus, and response. The graph notation for the metamodel nodes and arcs is shown in Figures 2-1a and 2-1b, respectively.

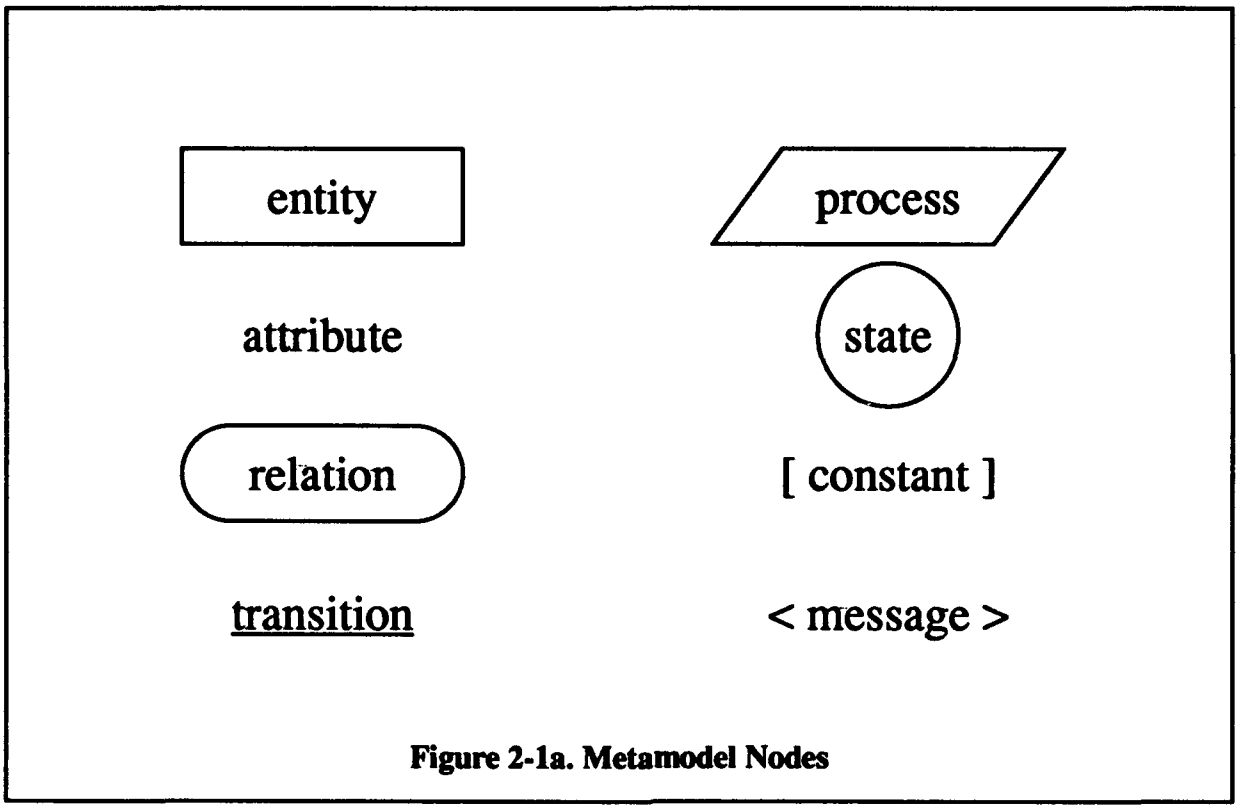


Figure 2-1a. Metamodel Nodes

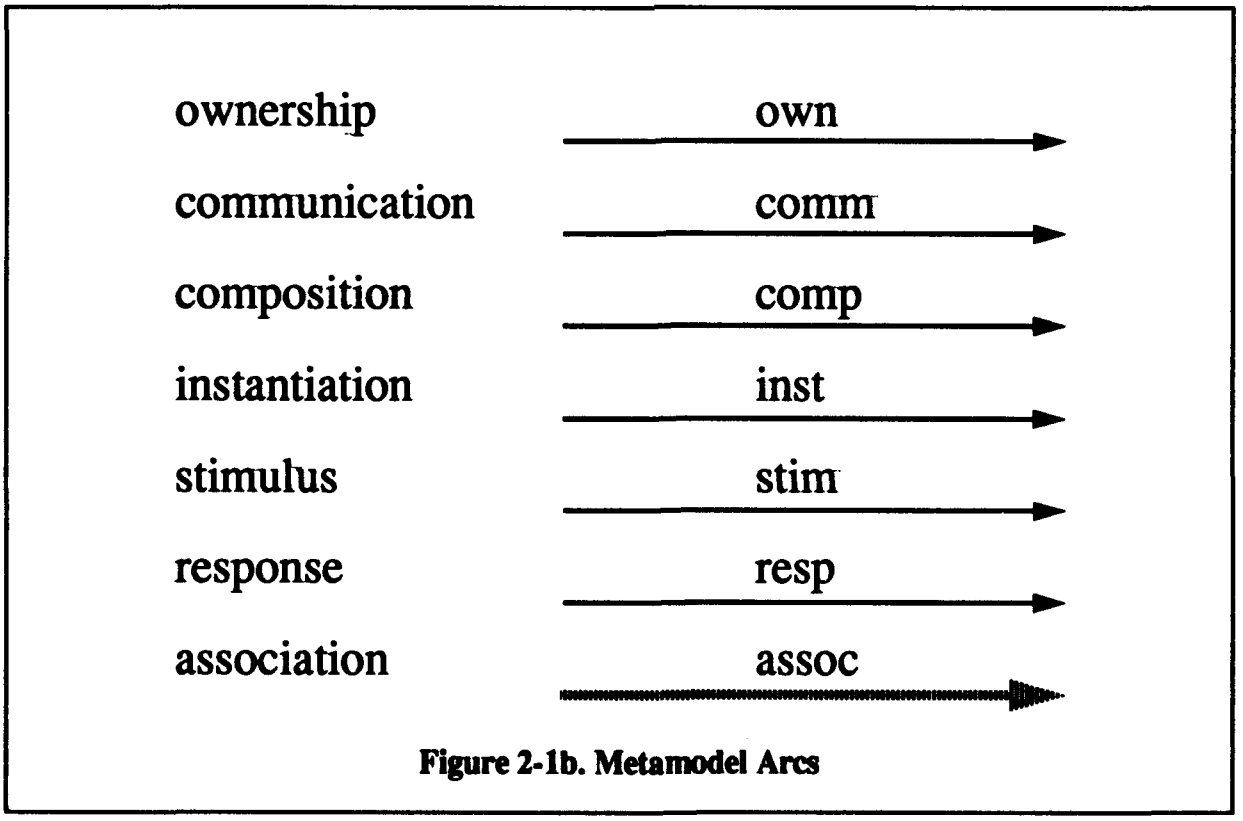


Figure 2-1b. Metamodel Arcs

Figures 2-2a and 2-2b depict examples of the node and arc notations. The items in Figure 2-2b mean (1) scale is one attribute owned by a map, (2) a user sends a new scale to adjust scale, (3) a legend is part of a map, (4) a road map is one instance of a map, (5) a map is associated in some unspecified way with a user, and (6) a map selection while in map select mode will display a new map.

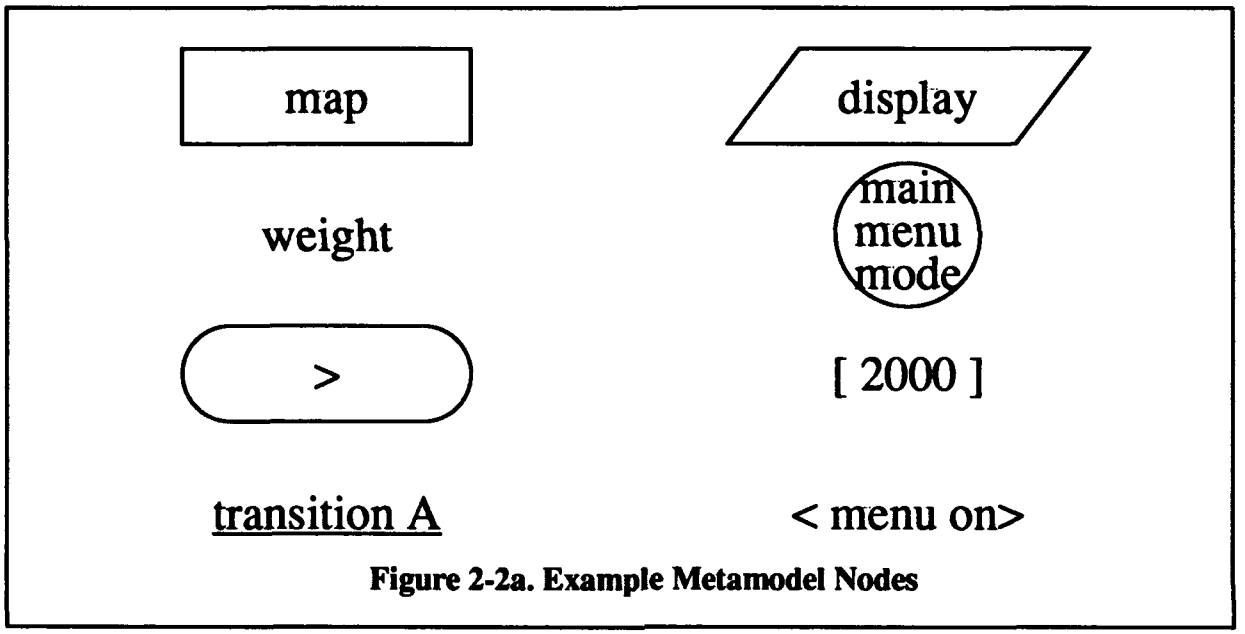


Figure 2-2a. Example Metamodel Nodes

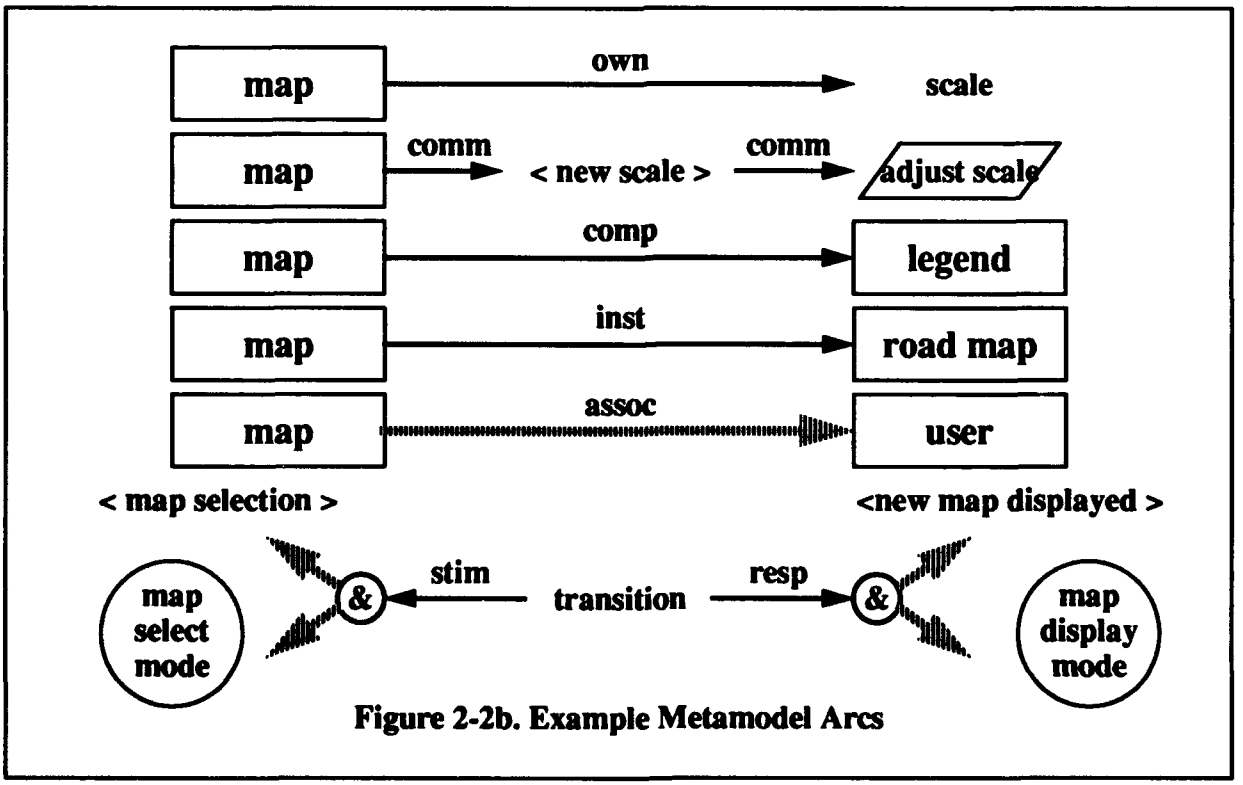


Figure 2-2b. Example Metamodel Arcs

2.3 Metamodel Nodes

In the following subsections, each of the node types is described and exemplified.

2.3.1 Entity Node

The *entity* node represents something physical in the real-world with well-defined boundaries. In both DFD and OOA models, they are representations of physical or conceptual entities in the "real world." In a DFD model, entities are represented as terminators, sometimes data stores, and sometimes data elements within data flows. The OOA object is not just an entity in the real world; it encapsulates attributes and services that act upon those attributes and upon the entity itself [COA91]. The DFD entities present a more diverse picture. The terminators and data stores are similar to the OOA objects in that they often represent physical entities in the real world, but they do not encapsulate attributes or services. Figure 2-3 shows some examples.

The entity node is the result of specifying any of the following:

Notation	Concept
DFD	Terminator, Data Store
OOA	Class, Object
Statechart	Orthogonal State

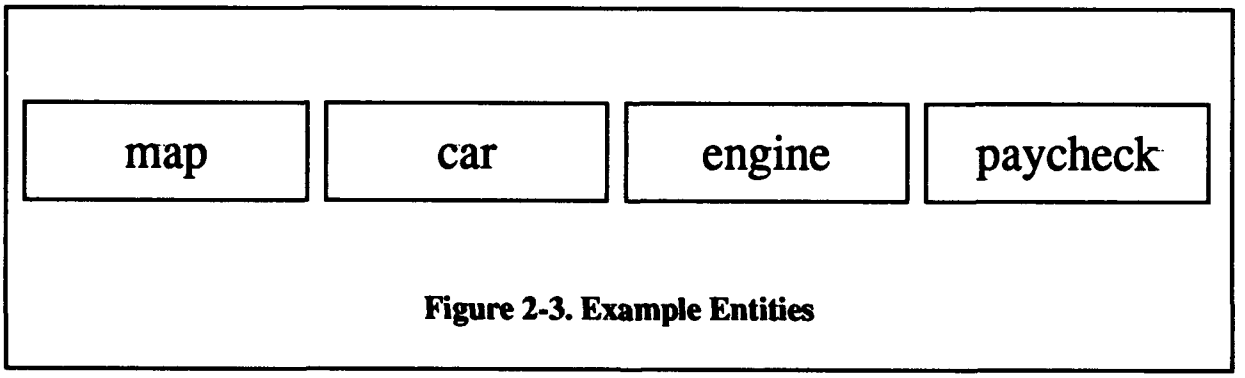


Figure 2-3. Example Entities

2.3.2 Process Node

The *process* represents some action or transformation of information in the requirements domain. For example, a process node might represent actions in the real world such as "deposit" and "withdrawal." This idea of process is broad and includes that of function (transform or mapping inputs to outputs); it includes that of activity (something performed) that potentially can produce an effect. In a DFD, process is exhibited as a transform. In the OOA model, process is exhibited as a service. One major difference between the DFD process and the OOA service is that the DFD process is not attached to anything else such as to an object in OOA. And while the DFD process may operate on values or attributes (represented as DFD data elements), it does not

explicitly belong to an object that possesses those attributes. The DFD process focuses almost exclusively upon the transformation, and this transformation could take effect upon attributes of several different entities (different OOA objects). Thus, a DFD process could represent the *combination* of OOA services that share similar transformational properties but are associated with different objects. Figure 2-4 shows some examples.

The process node is the result of specifying any of the following:

Notation	Concept
DeMarco DFD	Transform
OOA	Service
Ward DFD	Data Transform
Hatley DFD	Transform

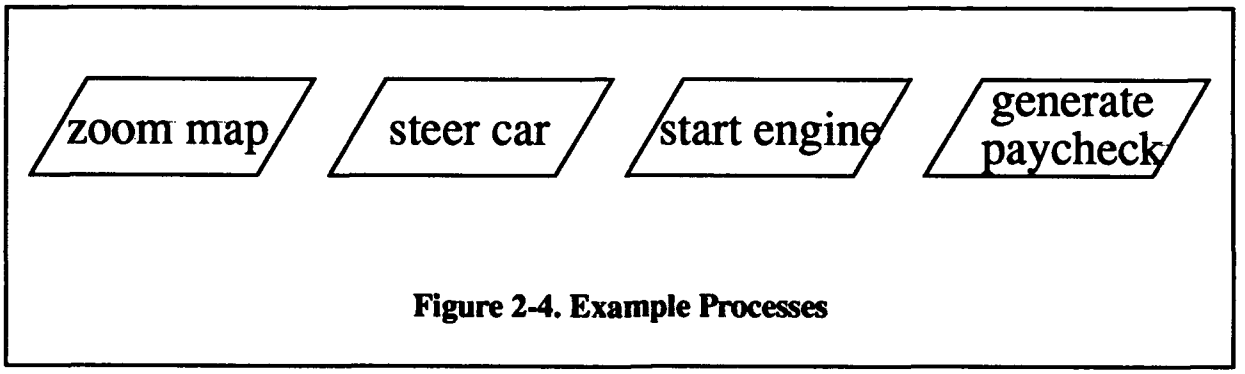


Figure 2-4. Example Processes

2.3.3 Attribute Node

The *attribute* node represents the assignment of a descriptive element or characteristic to another element. For example, an attribute node might represent attributes in the real world such as "customer name" or "account number." As descriptions or adjectives, attributes can have value. For example, an object could have the attribute of color, which in turn has the value of red. The value of attributes of a particular entity can tell us about its state. Of the nine notations used to derive the metamodel, only OOA represents attributes explicitly. Attributes in OOA provide the relevant traits of the OOA objects. Figure 2-5 shows some examples.

The attribute node is the result of specifying the following:

Notation	Concept
OOA	Attribute

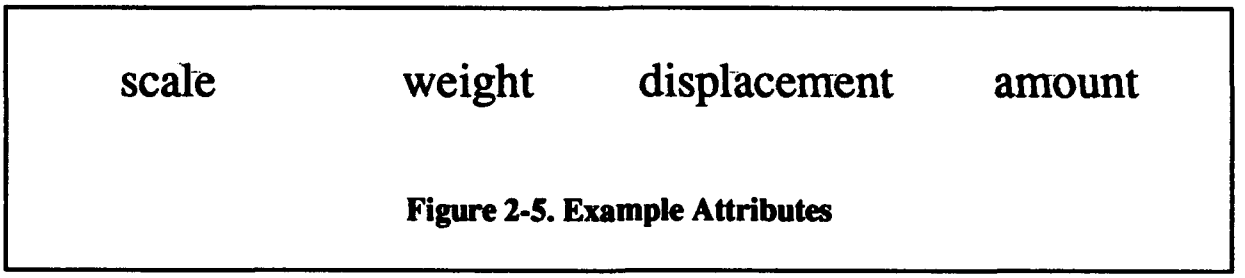


Figure 2-5. Example Attributes

2.3.4 State Node

The *state* node represents a set of values or a situation in the requirements world. Real systems are not purely reactionary; their behavior is not purely a function of the stimuli they receive. Instead, their behavior is a function of the stimuli they receive and the history of stimuli previously received. This history is recorded in a state. In DFDs, there is no concept equivalent to that represented by the state node. In OOA, an object could have a state that is expressed as the cross product of the values of the object's attributes, but no specific notation for state is used. Techniques that use DFDs (e.g., structured analysis/real-time [HAT87, WAR85, YOU89] and OOA diagrams (e.g., OOA [COA91]) recommend the addition of state transition diagrams or state-event-response tables to define the behavioral aspects of the system. A Petri net's state is the cross-product of the states of each of its places. A statechart's state is the cross-product of the states of each of its orthogonal states. Figure 2-6 shows some examples.

The state node is the result of specifying the following:

Notation	Concept
FSM	State
Ward	State
Hatley	State
Petri Nets	Place
Statechart	State

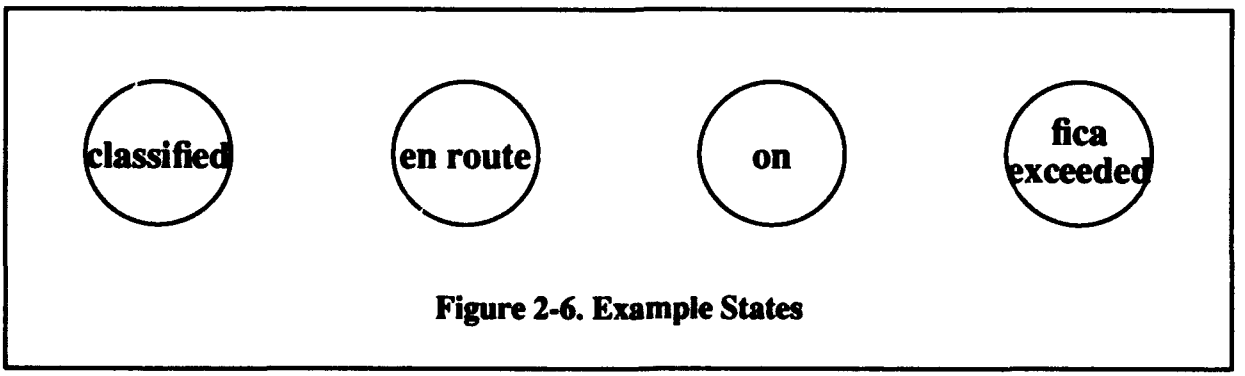


Figure 2-6. Example States

2.3.5 Relation Node

The *relation* node represents any relational and logical operator. Some examples are < (less than), <= (less than or equal), = (equal), > (greater than), >= (greater than or equal), AND, OR, NOT, and XOR, but other more application-specific relations can also be represented. The relation is used in conjunction with constants, attributes, messages, and other relations to establish a value or range of values. For example, a relation node might be used to represent the relation "less than" in the expression "less than minimum balance." Harel's statecharts allow Boolean expressions with relations to appear as conditions for state transition. Decision tables and trees allow Boolean expressions with relations to appear as condition row headers. Although not yet explored in detail, entity-relation diagrams will obviously make significant use of the relation node. Figure 2-7 shows some examples.

The relation node is the result of specifying the following:

Notation	Concept
Statechart	Conditional Transition with a Boolean Operator
Decision Table	Condition with a Boolean Operator
Decision Tree	Condition with a Boolean Operator

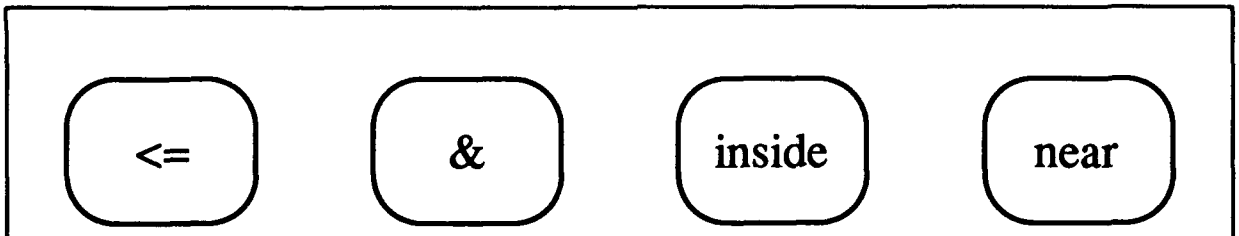


Figure 2-7. Example Relations

2.3.6 Constant Node

The *constant* node represents a specific value in the requirements domain. One interpretation of value is the assignment of worth to an attribute. Value can be assigned as a quality or as a quantity; it can be a single, discrete value or a range. In whatever form, some degree of worth, whether relative or absolute, can be assigned to an attribute. For example, a requirement in OOA may state that the weight of a vehicle may be between 1000 and 5000 pounds. To construct this range of values, it is necessary to use two constants (1000, 5000) and a logical operator (<). Thus, the allowable weight could be expressed as two Boolean expressions: (1000 pounds < allowable weight) and (allowable weight < 5000 pounds). The constant is used in conjunction with the relation node to define a value or range of values for an attribute. Figure 2-8 shows some examples.

The constant node can be the result of specifying the following:

Notation	Concept
Statechart	Conditional Transition with a Constant
Decision Table	Condition with a Constant
Decision Tree	Condition with a Constant

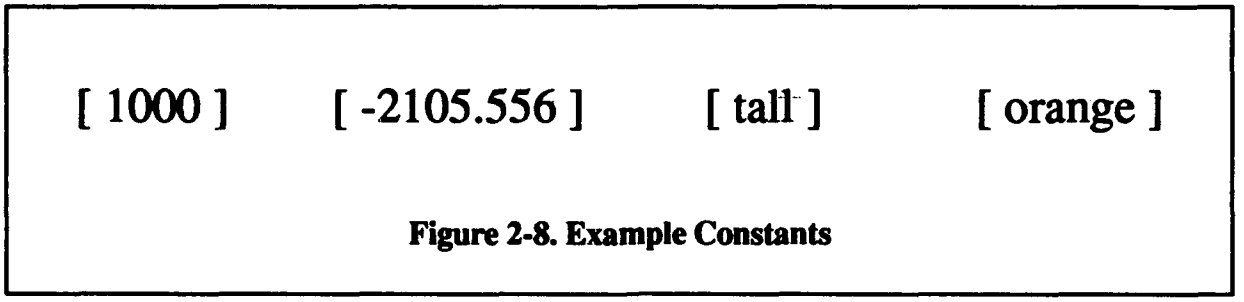


Figure 2-8. Example Constants

2.3.7 Transition Node

The *transition* node represents the change from a state to another state. For example, a transition node might represent the real world state transition from "account overdrawn" to "account above minimum balance." A state transition is fired by a combination of currently active states, a set of conditions, and the presence of various signals. The result of the firing is typically (a set of) new states, a (set of) new signal(s), and possibly a set of post-conditions. DFDs and OOA diagrams do not explicitly define state, let alone state transition. However, Ward, Hatley, Yourdon, and Coad have all suggested augmenting these with other notations that do capture state transition. FSMs, Petri nets, and statecharts explicitly define transitions between states. Decision tables and trees implicitly define transitions between states.

A transition node always has two arcs emanating from it: stimulus and response. The stimulus arc points to all those states, conditions, and signals that are necessary to trigger this transition. The response arc points to all those new states, conditions, and signals that will be triggered by two transitions, as shown in Figure 2-9. See Sections 2.4.5 and 2.4.6 for details on the semantics of the stimulus and response arcs.

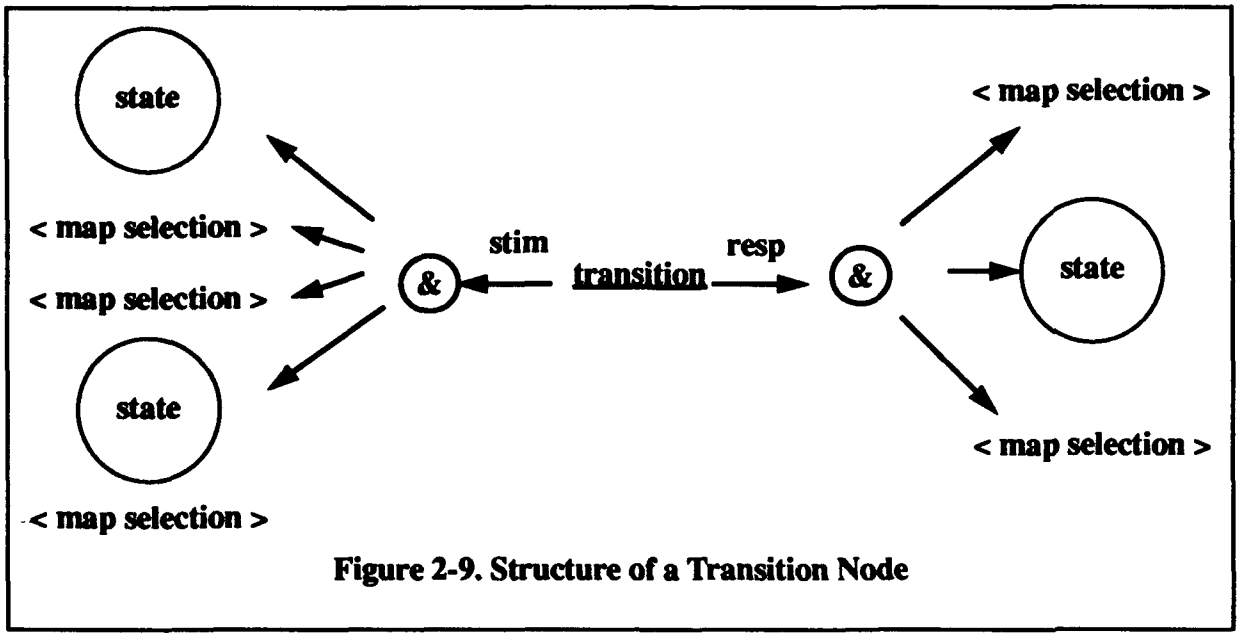


Figure 2-9. Structure of a Transition Node

2.3.8 Message Node

The *message* node represents something that is transferred (communicated) from one element to another. In DFDs, data elements move between the terminators, data stores, and processes. For example, a message node might represent the real world transfer of information such as "$1000" from the object "customer" to the process "credit customer's account." In OOA, objects have message connections whereby the objects request services of other objects. Messages may be either discrete or continuous, and own an attribute of DISCRETE or CONTINUOUS in the metamodel. They may also be data or control, in which case they own an attribute DATA or CONTROL in the metamodel.

The message node is the result of specifying the following:

Notation	Concept
DeMarco DFD	Data Flow
Ward DFD	Data Flow, Control Flow
Hatley DFD	Data Flow, Control Flow
OOA	Message Connection
FSM	Stimulus, Response
Petri net	Transition Label
Statechart	Stimulus, Response

2.4 Metamodel Arcs

In the following subsections, each of the arc types is described and exemplified.

2.4.1 Ownership Arc

The *ownership* arc represents the capability for one element to be a property or quality of another. For example, the OOA object (stored as an entity node) "vehicle" *owns* the attribute "vehicle number" (stored as an attribute node) where vehicle number is a quality or characteristic of vehicle. Another example in OOA is where the object vehicle owns the service "update vehicle number."

The ownership arc is the result of specifying the following:

Notation	Concept
OOA	Object with Attribute and/or Service
FSM	Start State
Statechart	Start State

The possible ownership relationships, by node, are the following:

Entity: An entity can own processes, states, attributes, and other entities.

Process: A process can own entities, states, and attributes.

State: A state can own attributes and processes.

Attribute: An attribute can own other attributes, relations and constants.

Transition: A transition may own attributes, such as a maximum response time.

Relation: None.

Constant: None.

Message: A message can own attributes, such as a range of acceptable values.

2.4.2 Communication Arc

The *communication* arc represents the flow of information from one element to another. It can be the result of specifying data flow or control flow in a DFD, or a message connection in OOA. In the OOA model, objects can be linked with "message connections," whereby one (sender) object requests a service of another (receiver) object. Communication in the metamodel is a three-way relation among the sender node (either entity or process), the message node, and the receiver node (entity or process). This is illustrated in the metamodel with two directed arcs: the first from the sender to the message and the second from the message to the receiver. For example, a communication arc might be used to link the message node "$1000" from the

(sending) "customer" entity to the (receiving) process "credit account." Communication arcs can be labeled as data, control, wake-up, or go-to-sleep. Data and control can be additionally labeled as either discrete or continuous.

The communication arc is the result of specifying the following:

Notation	Concept
DFD	Data Flow
Ward DFD	Data Flow, Control Flow
Hatley DFD	Data Flow, Control Flow
OOA	Message Connection

The possible communication relationships, by node, are the following:

Entity: An entity can have a communication relationship to (i.e., can "send") a message node.

Process: A process can have a communication relationship to (i.e., can "send") a message node.

State: In a Moore model of a finite state machine, a state can have a communication relationship to (i.e., can "send") a message node.

Attribute: None.

Transition: None.

Relation: None.

Constant: None.

Message: The message node can have a communication relationship with (i.e., can "be received by") an entity, state, or process node.

2.4.3 Composition Arc

The *composition* arc represents the capability to define constituent parts of an element. This concept appears in DFD models as the "leveling" in which a process can be partitioned into subprocesses [DEM79]. For example, the "update customer account" process might be composed of the processes "credit customer account" and "debit customer account." This concept appears in OOA models as the "Whole-Part" feature. An OOA object can have a "Whole-Part" relationship with other objects that together compose the higher-level object [COA91]. The concept of composition is not defined in FSMs models. The concept appears in statecharts [HAR87] when we decompose states into substates orthogonally.

The composition arc is the result of specifying the following:

Notation	Concept
DFD	Decompose Bubble into DFDs
Ward	Decompose Bubble into DFD
Hatley	Decompose Bubble into DFD/CFD
OOA	Whole-Part Structure
Statechart	"and" Decomposition

The possible composition relationships, by node, are the following:

Entity: An entity can be composed of subentities.

Process: A process can be composed of subprocesses.

State: A state can be composed of substates.

Attribute: An attribute can be composed of subattributes.

Transition: None.

Relation: None.

Constant: An aggregate constant can be composed of simpler constants.

Message: A message can be composed of submessages.

2.4.4 Instantiation Arc

The *instantiation* arc refers to the capability of one element inheriting properties from another. By inheriting properties, this element is an example or instance or more specific realization of the "parent" element. In OOA, this concept is represented with the "Gen-Spec" relationship between objects. The "Generalization-Specialization" allows the specialized objects to inherit attributes and services from the more general class or class-&-object. Such inheritance features are not explicitly present in DFDs or FSMs. The concept does appear in statecharts in which orthogonally decomposed states inherit transitions of their parent superstates. An instance of a node will share all the original node's attributes and relationships. The instantiation arc is directed from the original node to the instance. This relationship can exist only between nodes of like kind, e.g., between entities, processes, attributes, and messages. For example, the "customer" entity might have instances of "business" or "individual." These entities would inherit the attribute's "name" and "address."

The instantiation arc is the result of specifying the following:

Notation	Concept
OOA	Gen-Spec
Statechart	"or" Decomposition

The possible instantiation relationships, by node, are the following:

Entity: An entity can have instances of itself.

Process: A process can have instances of itself.

State: A state can have instances of itself.

Attribute: An attribute can have instances of itself.

Transition: None. Relation: A relation can have instances of itself.

Constant: None.

Message: A message can have instances of itself.

2.4.5 Stimulus Arc

The *stimulus* arc represents an external event, action, or condition that triggers a state transition. A stimulus arc can be directed toward a message, a state or a relation. When pointing to a message, the meaning is "Is this message present?" When pointing to a state, the meaning is "Is this state active?" When pointing to a relation, the meaning is "Is this relation true?" In FSMs, multiple signals and single states can serve as stimuli. In Petri nets, multiple states and single signals can serve as stimuli. In statecharts, multiple signals, multiple states, and multiple conditions can serve as stimuli. In decision tables and trees, multiple conditions can serve as stimuli. When signals, states or conditions are used as stimuli to a state transition they may either be consumed (i.e., disappear after the transition occurs) by the transition or remain persistent (i.e., remain after the transition occurs). Thus each stimulus relationship possesses an attribute of consumed or persistent. In FSMs all states and signals serving as stimuli are consumed, because the old state changes with the state transition and the stimulating signal serves its purpose (i.e., triggers the transition) but then is no longer perceivable by later activities. In statecharts, states from which state transition arcs emanate are always consumed (just like their FSM counterparts); state names appearing in brackets as a label on a transition arc are always persistent; conditions appearing in parentheses as a label on a transition arc are always persistent. In decision tables and trees, all stimuli are persistent, i.e., a column in a table becoming true does not immediately change any of the conditions that caused it to be true.

The possible stimulus relationships, by node, are the following:

Entity: None.

Process: None.

State: None.

Attribute: None.

Transition: A transition node can be linked via a stimulus arc to a message node, a state node, or a relation node.

Relation: None.

Constant: None.

Message: None.

2.4.6 Response Arc

The *response* arc represents an event, action, or condition that is triggered by a state transition. A response arc can be directed toward a message, a state or a relation. When pointing to a message, the meaning is "Generate this message." When pointing to a state, the meaning is "Make this state active." When pointing to a relation, the meaning is "This relation becomes true." In FSMs, multiple messages can serve as responses. In Petri nets, multiple states can serve as responses. In statecharts, multiple messages and multiple states can serve as responses. In decision tables and trees, multiple conditions, multiple states and multiple messages can serve as responses. A response arc may be the result of specifying a response in a finite state machine diagram, or an activate or deactivate control flow in Ward DFDs, or any control signal in Hatley DFDs.

The possible response relationships, by node, are the following:

Object: None.

Process: None.

State: None.

Attribute: None.

Transition: A transition node can be linked via a response arc to a message node, a state node, or a relation node.

Relation: None.

Constant: None.

Message: None.

2.4.7 Association Arc

The *association* arc represents an association or undefined relationship between two entities in the requirements domain. For example, an association arc might represent a link between two objects such as "customer" and "account database." Or the association arc might be used to create expressions linking relation, constant, and attribute nodes to create Boolean expressions such as "more than $50,000."

The possible association relationships, by node, are the following:

Entity:	An entity can be associated with other entities, processes, states, attributes, transitions, and messages.
Process:	A process can be associated with entities, other processes, states, attributes, transitions, and messages.
State:	A state can be associated with attributes, other states, and messages.
Attribute:	An attribute can be associated with entities, processes, states, and relations.
Transition:	None.
Relation:	The relation can have an association relationship with an attribute or with a constant.
Constant:	The constant can have an association relationship with a relation node.
Message:	None.

SECTION 3

3. Mappings to the Requirements Techniques

To verify that the metamodel is valid, it is necessary to establish a set of mappings to and from the set of requirements notations. These same mappings will be used by a requirements engineering environment to translate a specification into the internal metamodel representation and back into any desired notation.

3.1 DeMarco Data Flow Diagram

3.1.1 Overview

DeMarco data flow diagrams (DFD) describe the movement and transformation of data throughout a system, highlighting the system processes and the data flows between processes. This notation is part of a larger methodology called Structured Analysis [DEM79]. The major components of the DFD are the process, terminator, data store, and data flow. DeMarco [DEM79] defines these components:

Process:	Transformation of input data flow(s) into output data flow(s). Shown by the circles, or "bubbles."
Data Flow:	A pipeline along which information of known composition is passed. Shown by curved, directed arrows.
Data store:	A repository of data; a time-delayed data flow. Shown by two parallel lines.
Data Source and Sink:	A person or organization, lying outside the context of a system, that is a net originator or receiver of system data. Shown by a box.

Processes are the functions to be performed by the system being specified. Data flows are the connections between the system functions. Data stores show collections (aggregates) of data that the system must remember for a period of time. Data sources and sinks (also called terminators) are the external entities with which the system communicates, such as individuals, groups of people, external computer systems, or organizations. Later in this document, data flow diagrams will be expanded to incorporate Ward's, and Hatley's extensions.

Figure 3-1 illustrates the notation for the DFD. A context diagram shows only one bubble, the system, and shows the terminators that send input to and receive output from the system. The system bubble is then refined into more detailed DFDs. There are several requirements techniques that utilize DFDs, with the more commonly known developed by Yourdon [YOU89], DeMarco [DEM79], Gane and Sarson [GAN79], Ward [WAR85], and Hatley [HAT87]. Most of these techniques also include a data dictionary and a set of process specifications. The data dictionary clarifies the definitions of data elements in the DFD. Each data element should be identified and defined along with any constituent parts of the data element. Specifically, each data element should have a meaning, composition, and set of allowable values [YOU89]. The process

specification describes the internal design of the bottom-level, primitive processes in a DFD. At this level, the specification describes design and algorithms for the system implementation and hence, is beyond the requirements domain.

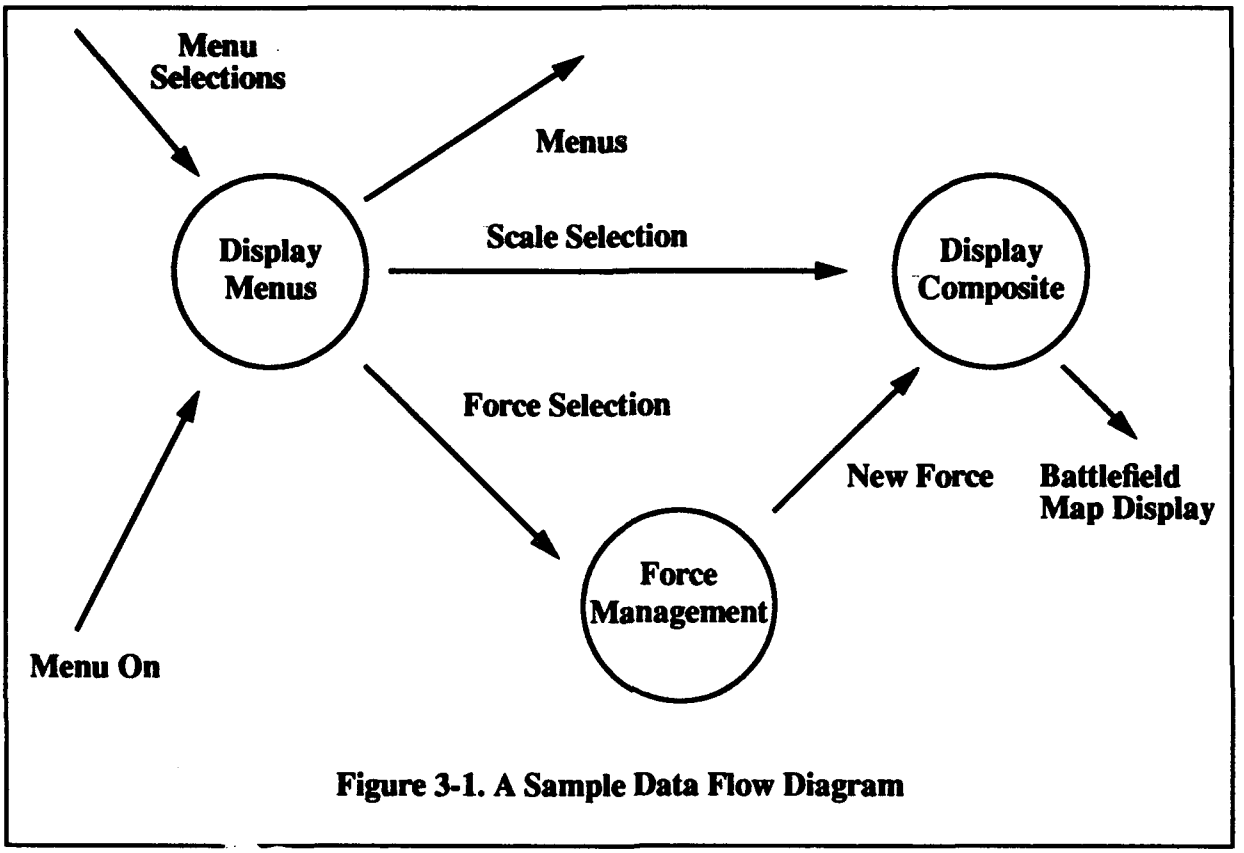


Figure 3-1. A Sample Data Flow Diagram

3.1.2 Mapping Data Flow Diagram to Metamodel

The mapping from the DeMarco DFD to the metamodel is illustrated in Figure 3-2. DFD transforms or processes are represented as processes in the metamodel. It is assumed that all data flows are messages until other requirements notations specify them otherwise (e.g., as attribute, class, or constant). DFD terminators and data stores are both represented as entities in the metamodel. When a DFD is refined into a more detailed DFD, all elements (i.e., terminators, processes, data stores) in that DFD are recorded in the metamodel as components of the parent DFD using composition arcs.

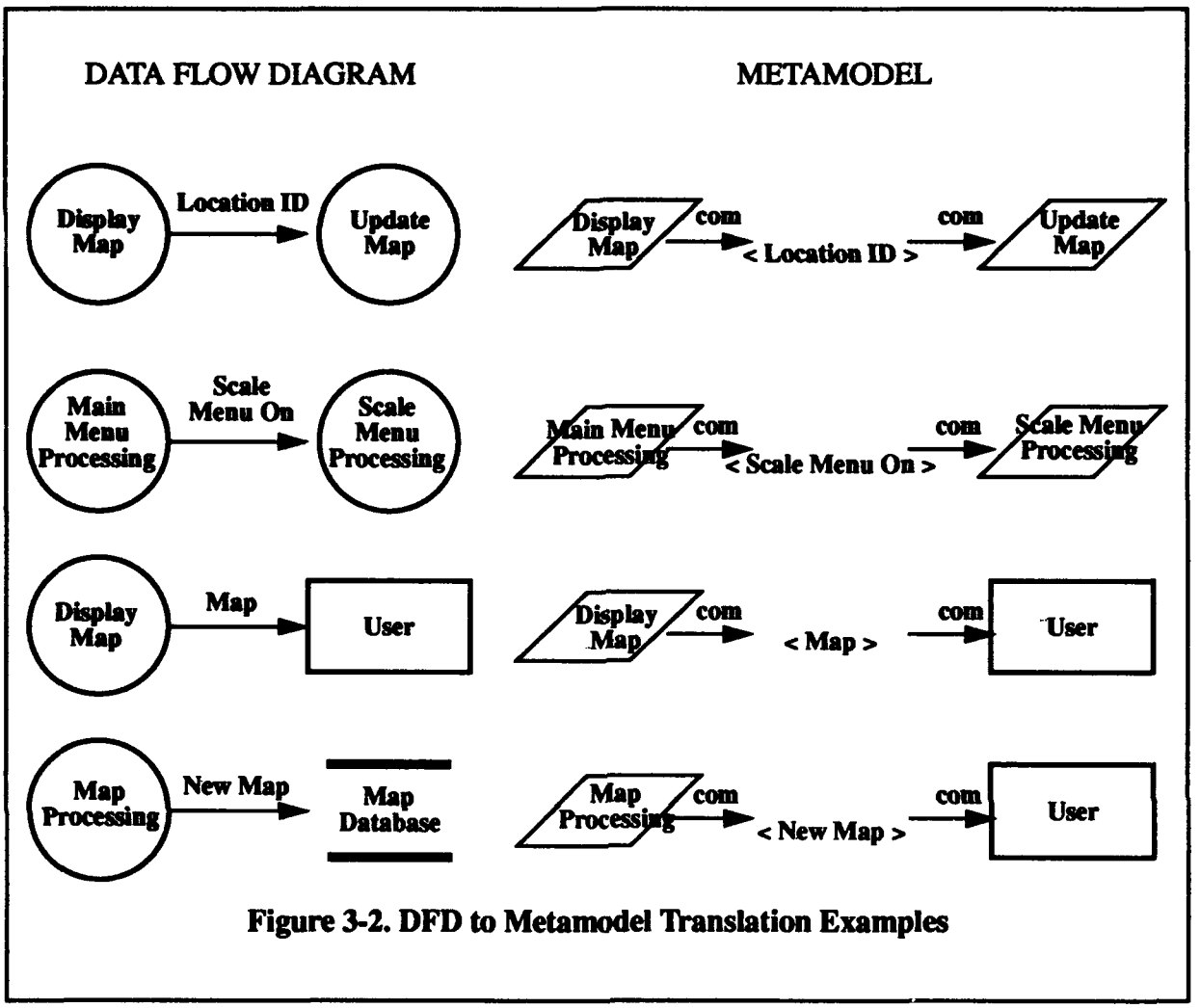


Figure 3-2. DFD to Metamodel Translation Examples

3.1.3 Mapping Metamodel to Data Flow Diagram

In translating from the metamodel to the DFD (Table 3.1), all metamodel processes map to DFD processes (bubbles). All messages between processes map to data flows. Any entity whose attribute "external/internal" indicates that it is "external" maps to a terminator. If the entity is "internal," then it maps to a data store. If a metamodel process has a composition relationship with other elements, then the lower level elements become part of the next-level-down DFD.

Table 3.1 Mapping from Metamodel to Data Flow Diagram

Metamodel	Data Flow Diagram
Entity	Data Store, Terminator
Process	Process
Message Between Entities and/or Processes	Name of Data Flow
Attribute	N/A
State	N/A
Transition	N/A
Relation	N/A
Constant	N/A
Ownership	N/A
Association	N/A
Composition Among Processes	Next Level DFD
Composition Among Other Nodes	N/A
Instantiation	N/A
Communication	Data Flow
Stimulus	N/A
Response	N/A

3.2 Ward Structured Analysis

3.2.1 Overview

The Ward DFD notation extends DeMarco's original DFD notation by adding control processes and control flows [WAR85]:

Control process: This type of process coordinates the activities of other processes. Only control flows make up the inputs and outputs to the control process. Shown as a dashed circle.

Control flow: This type of flow represents a signal or interrupt that is sent to "wake up" the receiving process. According to Yourdon [YOU89], this type of signal is binary (on or off) and does not contain data with value. Shown as a dashed directed arc.

There are three types of control flows:

Signal:	Reporting an event.
Activation:	A direct overt action to initiate another process.
Deactivation:	A direct overt action to stop another process.

There are also two types of data flows:

Discrete:	A single item of data.
Continuous:	A source of constantly available and perhaps continuously changing data.

Figure 3-3 shows a sample Ward-style data flow diagram. Ward also added finite state machines as a means of specifying behavior of a process. These are described in Section 3.5.

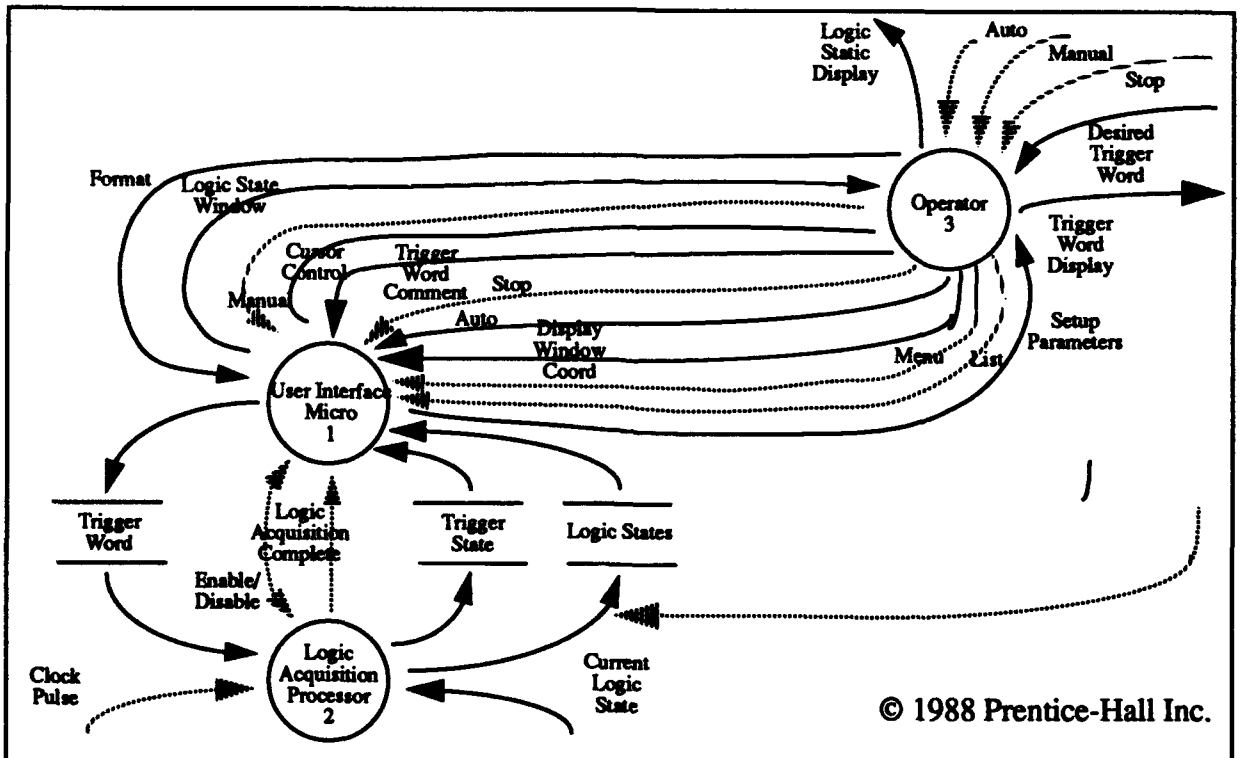


Figure 3-3. A Sample Ward Data Flow Diagram

3.2.2 Mapping from Ward to Metamodel

The mapping from the Ward DFD to the metamodel is illustrated in Figure 3-4. DFD transforms are represented as processes in the metamodel. It is assumed that all data and control flows are messages until other requirements notations specify them otherwise (e.g., as attribute,

class, or constant). Signal control flows are mapped into discrete control event signals. Activation control flows are mapped into discrete control wake-up signals. Deactivation control flows are mapped into discrete control go-to-sleep signals. DFD terminators and data stores are both represented as entities in the metamodel. When a DFD is refined into a more detailed DFD, all elements (i.e., terminators, processes, data stores) in that DFD are recorded in the metamodel as components of the parent DFD using composition arcs.

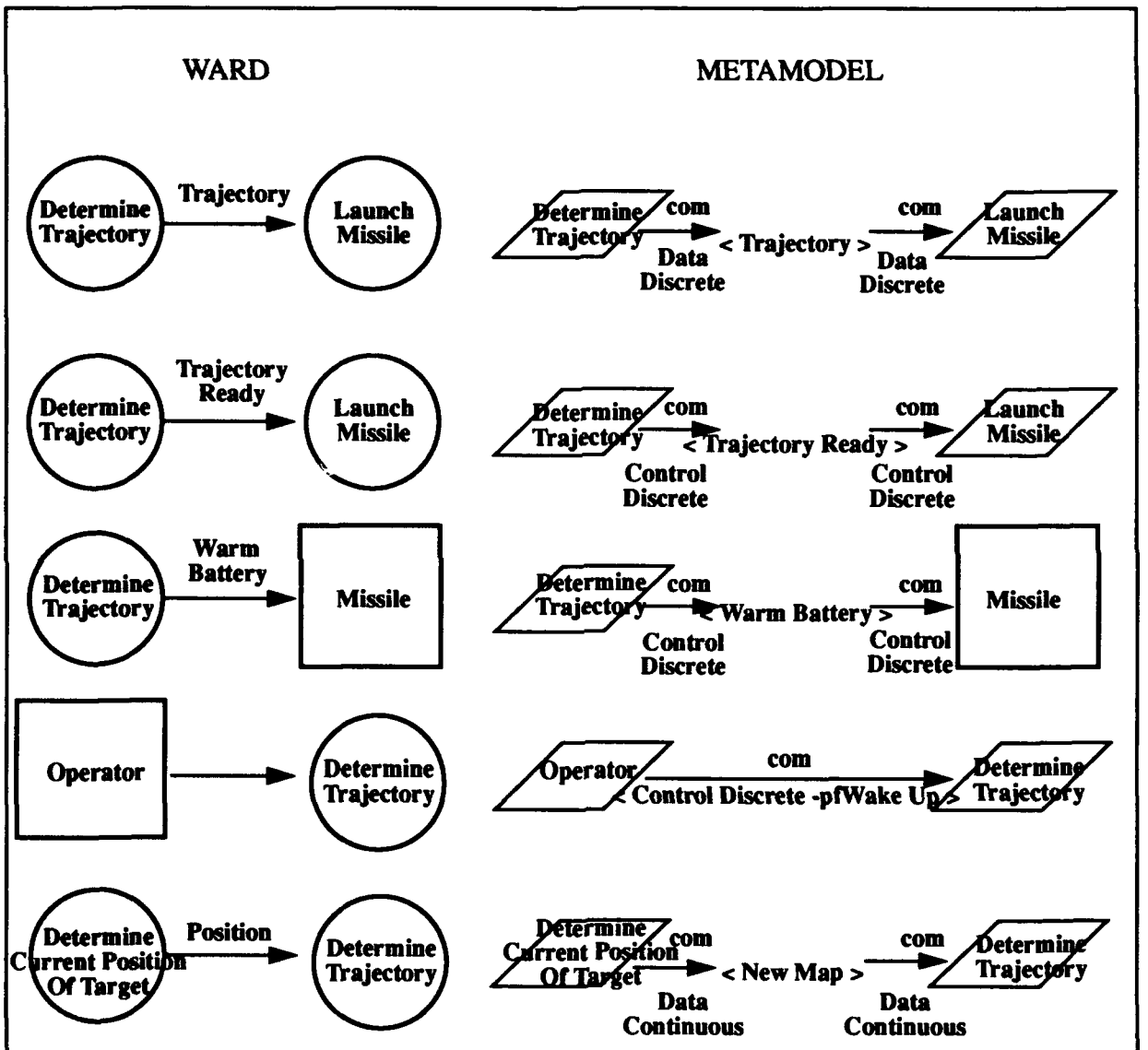


Figure 3-4. Ward to Metamodel Translation Examples

3.2.3 Mapping Metamodel to Ward

In translating from the metamodel to Ward (Table 3-2), all metamodel processes map to DFD processes (bubbles). All messages between processes map to their corresponding data or

control signals. Any entity whose attribute "external/internal" indicates that it is "external" maps to a terminator. If the entity is "internal," then it maps to a data store. If a metamodel process has a composition relationship with other elements, then the lower level elements become part of the next-level-down DFD.

Table 3.2 Mapping from Metamodel to Ward

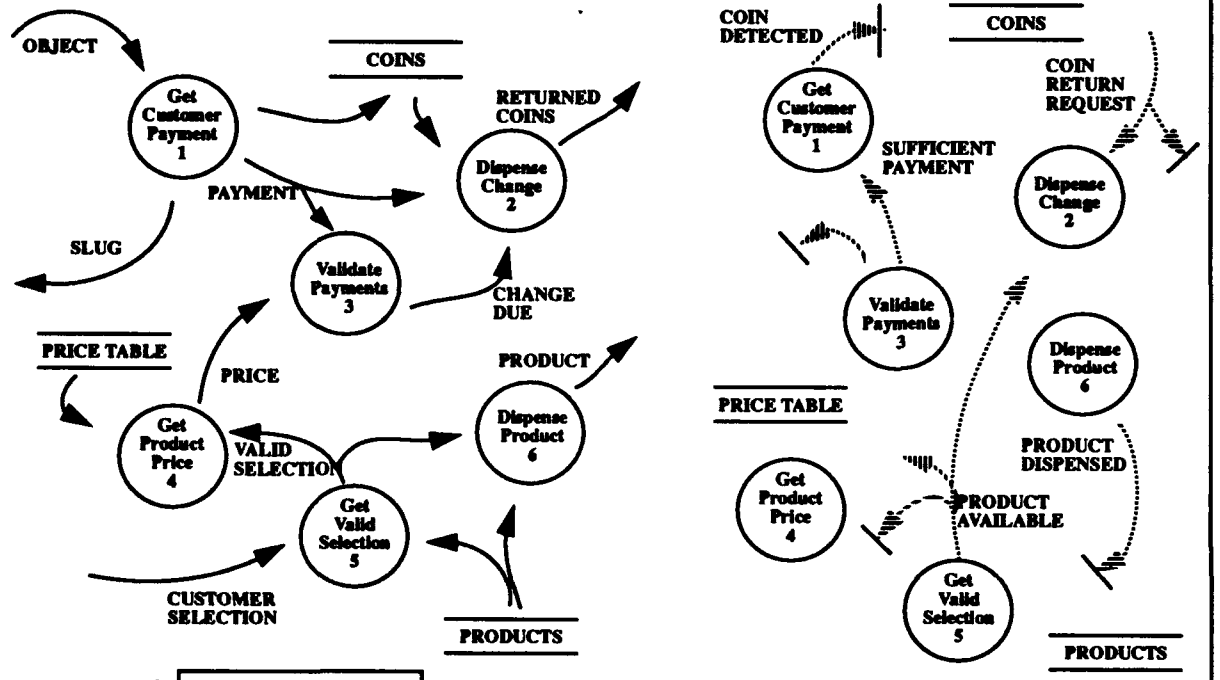
Metamodel	Ward
Entity	Data Store, Terminator
Process	Process
Message Between Entities and/or Processes	Name of Flow
Attribute	N/A
State	State
Transition	Transition
Relation	N/A
Constant	N/A
Ownership	N/A
Association	N/A
Composition of Process	Next Level DFD or FSM
Composition from Other Nodes	N/A
Instantiation	N/A
Communication (Data)	Data Flow
Communication (Control)	Control Flow
Stimulus (Consumed)	Stimulus on Transition
Stimulus (Persistent)	N/A
Response	Response on Transition

3.3 Hatley Structured Analysis

3.3.1 Overview

Hatley [HAT87] has extended DeMarco's notation in different ways. In particular, his DFD, data dictionary, and process specification are the same as DeMarco's, but he has added a control flow diagram (CFD), control specification (finite state machine) and process activation table to describe the external behavior of the system. In a CFD, control flows can terminate at another process, in which case that process is activated. Control flows can also terminate at bars,

in which case the control flow acts as a stimulus to the finite state machine. In response to the arrival of the stimulus, the finite state machine may change state and/or generate an internal signal. This internal signal then drives a process activation table, which specifies which processes (on the DFD and CFD) are activated. Figure 3-5 illustrates a sample Hatley series of diagrams.



Control Action \ Process Activated	Dispense Change	Dispense Product	Get Valid Selection
Accept Customer Request	0	0	1
Return Payment	1	0	0
Accept New Coin	0	0	0
Dispense Product	1	1	0

Figure 3-5. A Sample Hatley Model

3.3.2 Mapping from Hatley to Metamodel

The mappings from the Hatley to the metamodel are shown in Figure 3-6. Terminators map to metamodel entities with an "external" attribute; data stores to entities with an "internal" attribute. Hatley processes map to metamodel processes each with attributes as to control or data. Control flows become messages with the communication link possessing a control attribute. Control flows directed to a bar are considered stimuli in the Hatley notation and the metamodel. The finite state machine that goes with this control flow stimulus is transformed into the metamodel following the finite state machine mapping described in Section 3.5 of this paper.

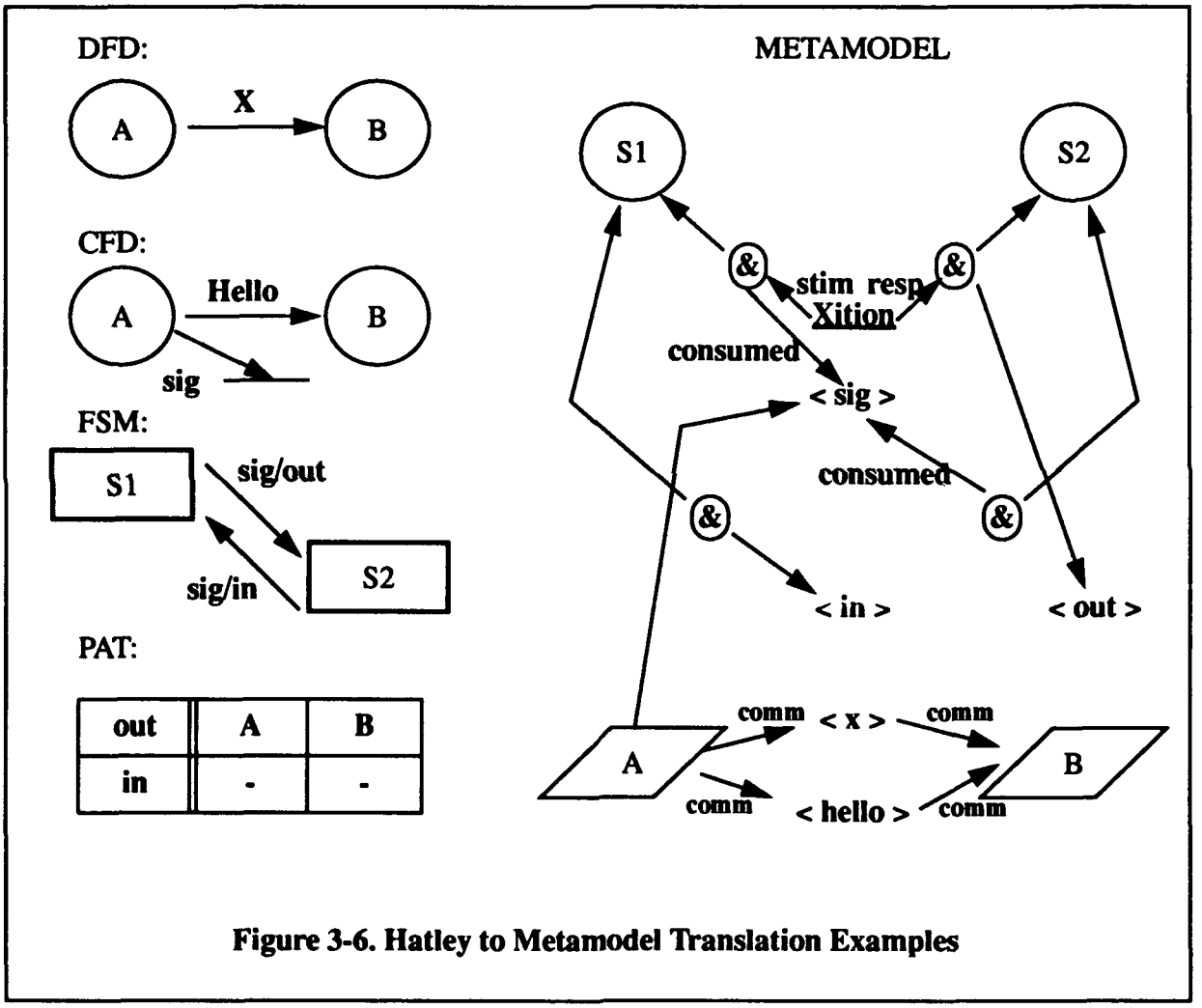


Figure 3-6. Hatley to Metamodel Translation Examples

3.3.3 Mapping Metamodel to Hatley

In translating from the metamodel to Hatley notations (Table 3-3), all metamodel processes map to DFD processes (bubbles). All messages between processes map to data signals. All control flows map to discrete control wake-up signals. Any entity whose attribute "external/ internal" indicates that it is "external" maps to a terminator. If the entity is "internal," then it maps to a data store. If a metamodel process has a composition relationship with other elements, then

the lower level elements become part of the next-level-down DFD or finite state machine.

Table 3.3 Mapping from Metamodel to Hatley

Metamodel	Hatley
Entity	Data Store, Terminator
Process	Process
Message Between Entities and/or Processes	Name of Flow
Attribute	N/A
State	State
Transition	Transition
Relation	N/A
Constant	N/A
Ownership	N/A
Association	N/A
Composition from Process	Next Level DFD or FSM
Composition from Other Nodes	N/A
Instantiation	N/A
Communication (Data)	Data Flow
Communication (Control)	Control Flow
Stimulus (Consumed)	Stimulus on Transition
Stimulus (Persistent)	N/A
Response	Response on Transition

3.4 Coad Object-Oriented Analysis

3.4.1 Overview

Coad's Object-Oriented Analysis (OOA) [COA91] emphasizes system objects. This notation shows the relationships between objects such as instantiation, composition, and cardinality. The major components of the OOA notation are:

Object: "An abstraction of something in a problem, reflecting the capabilities of a system to keep information about it, interact with it, or both; an encapsulation of Attribute values and their exclusive Services."

Class:	"A description of one or more Objects with a uniform set of Attributes and Services, including a description of how to create new Objects in the Class."
Class-&-Object:	"A term meaning 'a Class and the Objects in that Class.'"
Attribute	"An Attribute is some data (state information) for which each Object in a Class has its own value."
Service	"A Service is a specific behavior that an Object is responsible for exhibiting."
Message Connection	"A Message Connection models the processing dependency of an Object, indicating a need for Services in order to fulfill its responsibilities."
Instance Connection	"An Instance Connection is a model of problem domain mapping(s) that an Object needs with other Objects, in order to fulfill its responsibilities."
Gen-Spec Structure	A mechanism for distinguishing between classes of objects and their specific instances.
Whole-Part Structure	The Whole-Part Structure is a mechanism for defining an object or class along with its constituent parts.
Subject	"A Subject is a mechanism for guiding a reader... through a large, complex model. They define the top level of abstraction...."

Figure 3-7 shows a sample OOA diagram.

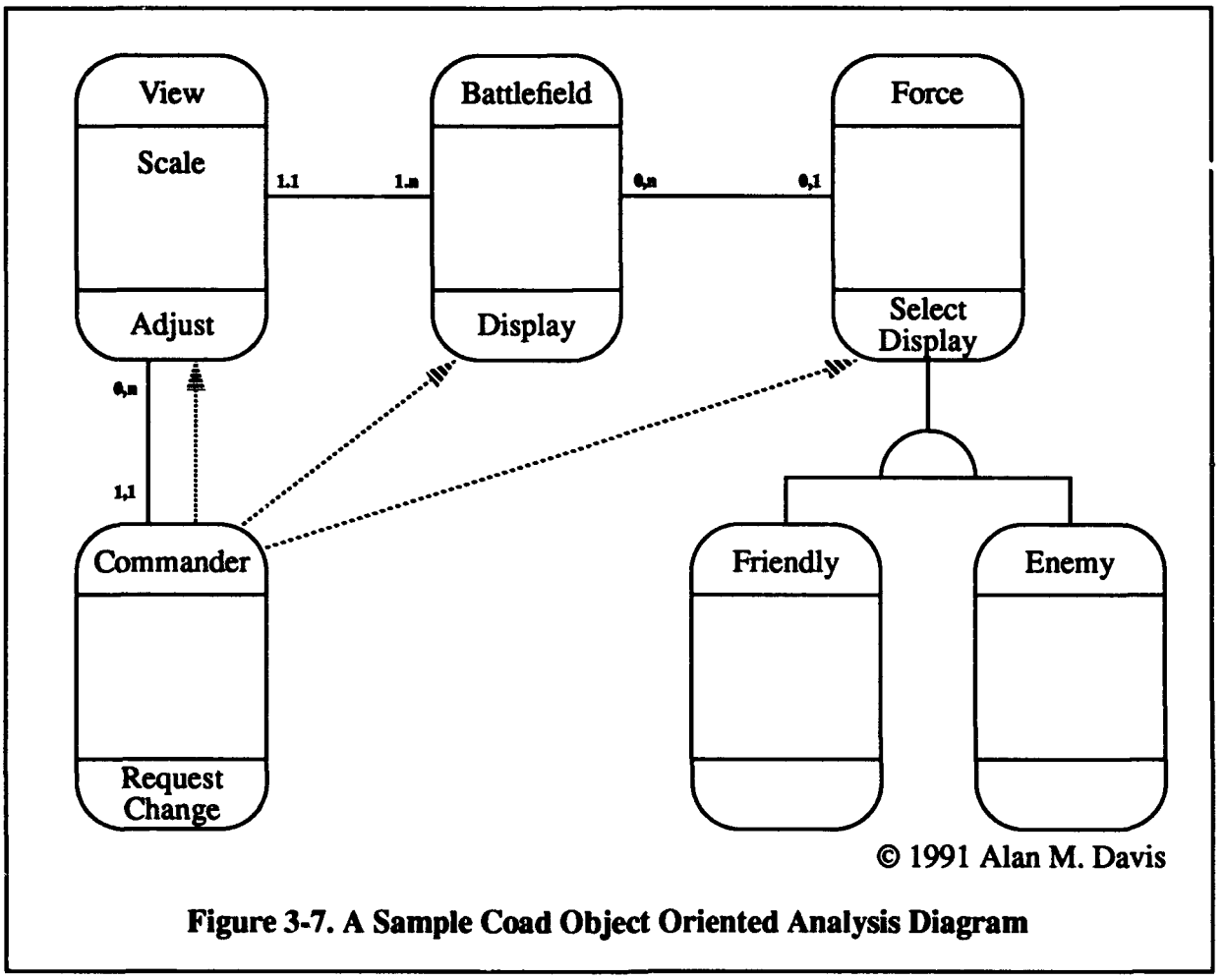


Figure 3-7. A Sample Coad Object Oriented Analysis Diagram

3.4.2 Mapping Object-Oriented Analysis to Metamodel

The mapping from OOA to the metamodel is illustrated in Figure 3-8. Subject, class, class-&-object, and object in OOA become entities in the metamodel. OOA attributes become attributes in the metamodel and since OOA attributes are associated with a particular OOA object, they are connected to the corresponding metamodel entity via an ownership arc. Likewise, OOA services become processes that are owned by the corresponding entity in the metamodel. An OOA instance connection between objects becomes two association arcs between the corresponding metamodel entities, one for each direction implied by the instance connection. The Gen-Spec relationship in OOA is represented by the instantiation arc directed to the more specific "instances" of the entity. The OOA Whole-Part relationship is represented by the composition arc which is directed from the "whole" entity to its constituent subentities. Message connections are represented by communication arcs between entities.

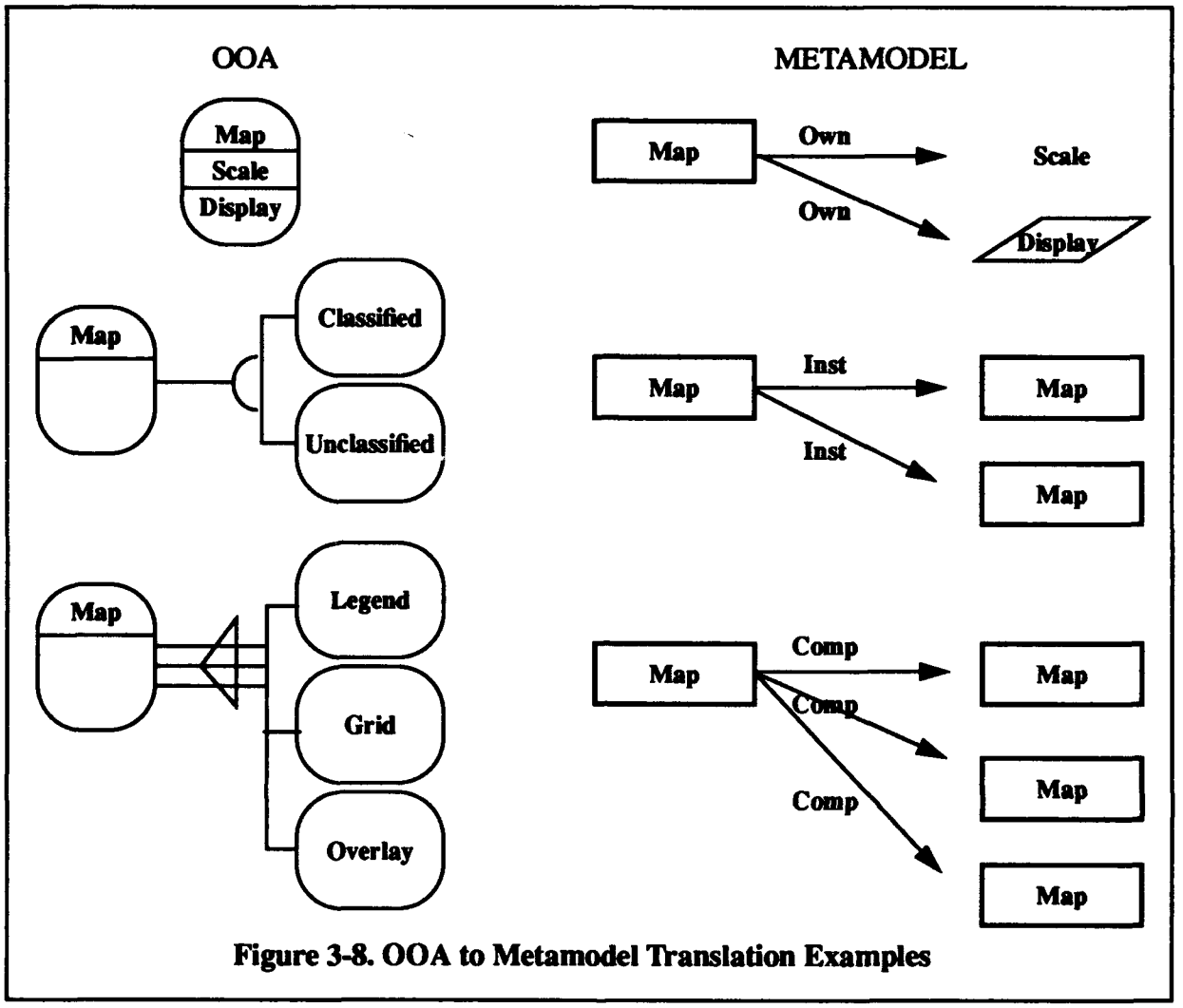


Figure 3-8. OOA to Metamodel Translation Examples

3.4.3 Mapping Metamodel to Object-Oriented Analysis

The mapping from the metamodel is summarized in Table 3-4. A metamodel entity that owns attributes and services becomes an OOA class-&-object until further information refines it. Processes owned by entities become services of the corresponding OOA class-&-object. Likewise, metamodel attributes become OOA attributes of the OOA class-&-object. If a metamodel entity exists without any attributes or processes attached to it, it is likely that this entity did not originate from an OOA specification but probably from another notation. In this case, the metamodel entity becomes an OOA object without attributes or services. However, the OOA model should be reviewed as to whether the entity is an OOA object or attribute.

Table 3.4 Mapping from Metamodel to Object Oriented Analysis

Metamodel	OOA
Entity	Class-&-Object, Class, Object
Process	Service
Message Between Entities and/or Processes	N/A
Attribute	Attribute
State	N/A
Transition	N/A
Relation	N/A
Constant	N/A
Entity Ownership of Attributes	Attribute in Object
Entity Ownership of Processes	Service in Object
Other Ownerships	N/A
Association Among Entities	Instance Connection
Association Among Other Nodes	N/A
Composition Among Entities	Whole-Part Connection
Composition Among Other Nodes	N/A
Instantiation Among Entities	Gen-Spec Connection
Instantiation Among Other Nodes	N/A
Communication	Message Connection
Stimulus	N/A
Response	N/A

3.5 Finite State Machines

3.5.1 Overview

Finite state machines can be used to describe a system's behavior. It is a hypothetical machine that may be in any one of a finite number of defined states (it starts in the START state) and that changes state and emits a response as functions of its current state and an incoming signal. In addition, every finite state machine has a start state. In a Mealy model, the response resulting from an arrival of a stimulus is associated with a state transition; in a Moore model, the

response is associated with a state. Figure 3-9 shows part of a Mealy model finite state machine.

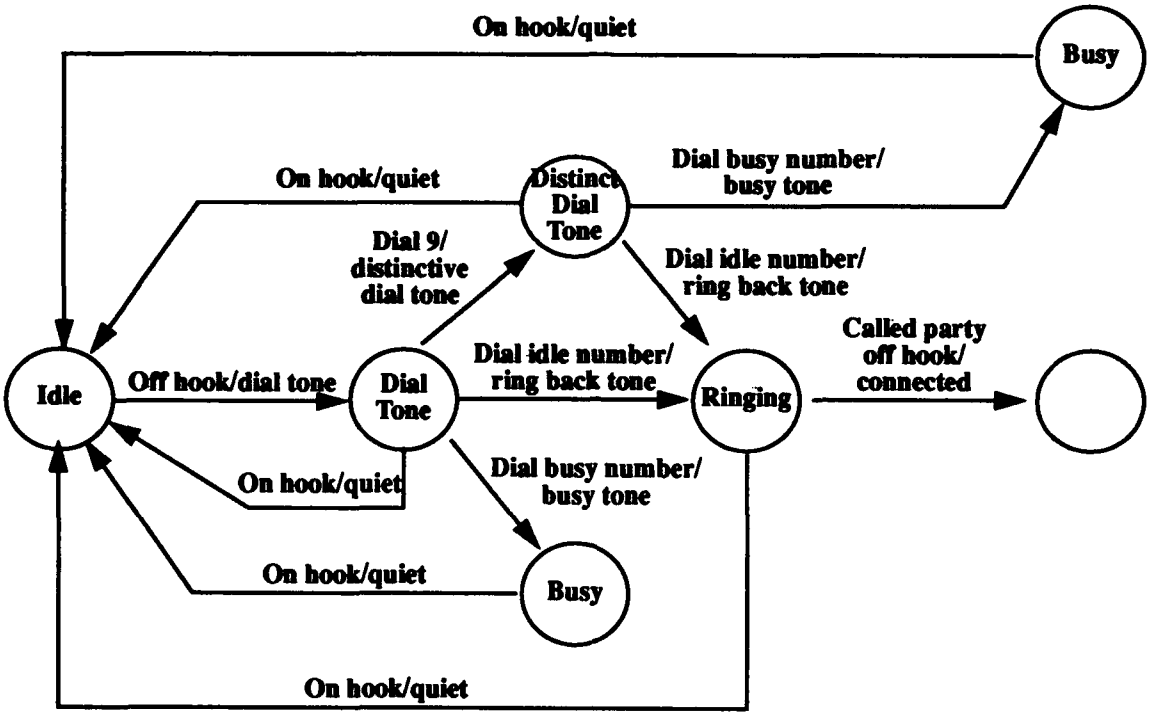


Figure 3-9. A Sample Mealy Finite State Machine

3.5.2 Mapping Finite State Machine to Metamodel

The mapping from the finite state machine is illustrated in Figure 3-10. FSM states becomes states in the metamodel. FSM state transitions become metamodel transition nodes that are linked via a stimulus arc to the originating state and to the stimulus; and via a response arc to the response and to the destination state. An FSM stimulus becomes a metamodel message that is linked via a stimulus arc from the transition node. An FSM response becomes a metamodel message that is linked via a response arc from the transition node. The start state is shown with a "Start" attribute owned by the start state.

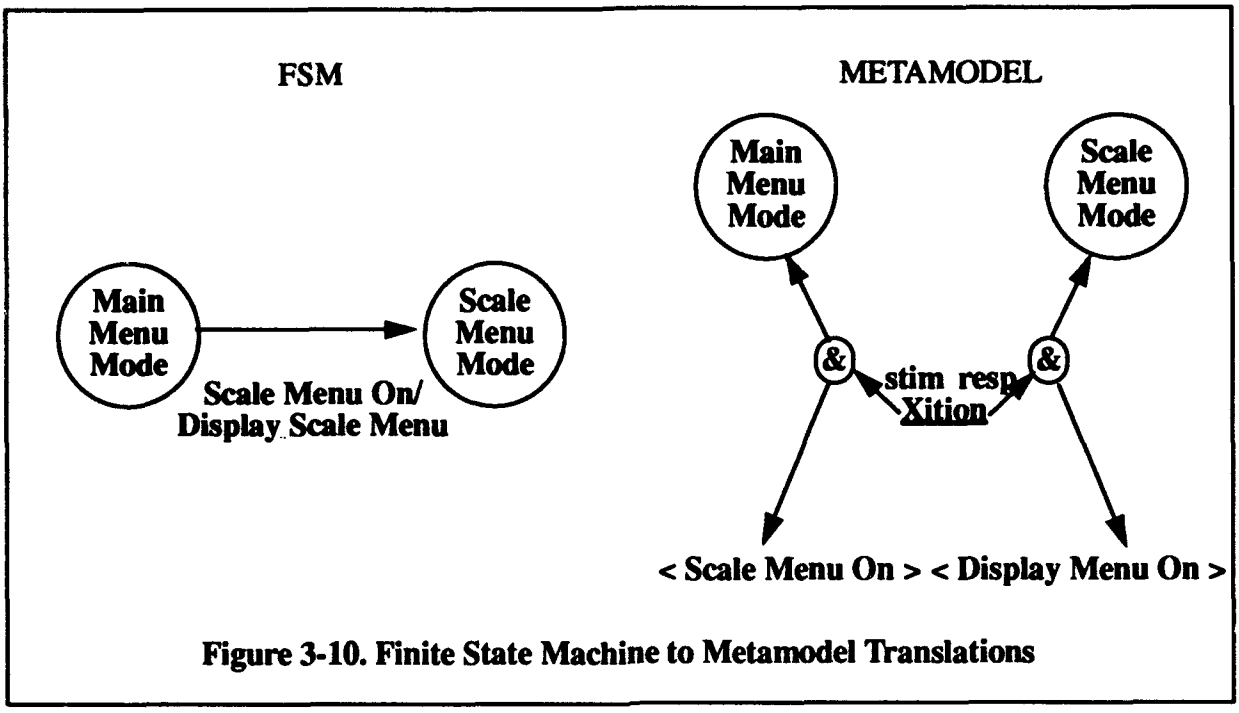


Figure 3-10. Finite State Machine to Metamodel Translations

3.5.3 Mapping Metamodel to Finite State Machine

The mapping from the metamodel to the finite state machine is summarized in Table 3-5. A metamodel state becomes an FSM state. Transition nodes become FSM transition arcs. Metamodel messages connected to the transition with a stimulus arc become FSM stimuli, and messages connected to the transition with a response arc become FSM responses.

Table 3.5 Mapping from Metamodel to Finite State Machine

Metamodel	Finite State Machine
Entity	N/A
Process	N/A
Message Between Entities and/or Processes	N/A
Attribute	N/A
State	State
Transition	Transition
Relation	N/A
Constant	N/A
State	Start State
Ownership of Attribute "Start"	N/A
Other Ownership	N/A
Association	N/A
Composition	N/A
Instantiation	N/A
Communication	N/A
Stimulus (consumed)	Stimulus
Stimulus (persistent)	N/A
Response	Response

3.6 Harel Statecharts

3.6.1 Overview

Harel statecharts are extensions to finite state machines. The Harel notation allows for states to be grouped as instances ("or" states) or components ("and" states) of a higher-level state. State transitions can also be dependent on the satisfaction of conditions and not just on the receipt of a stimulus. The following are the extensions to finite state machines defined by statecharts:

Or states — A state can be expanded into a set of less complex substates. A system being in the state means that it must be in one and only

	one of the substates. Shown as a standard finite state diagram within the bubble of a state.
And states	A state can be expanded into a set of less complex substates. A system being in the state means that it is in all of its substates as well. Shown as areas separated by dotted lines within a state.
Conditions	A transition may be dependent on a particular condition being true. Shown as a Boolean expression in parenthesis labeling a transition arc.
State Active	A transition may be dependent on a particular state being active. Shown as a state name in square brackets labeling a transition arc.

Figure 3-11 shows a typical statechart.

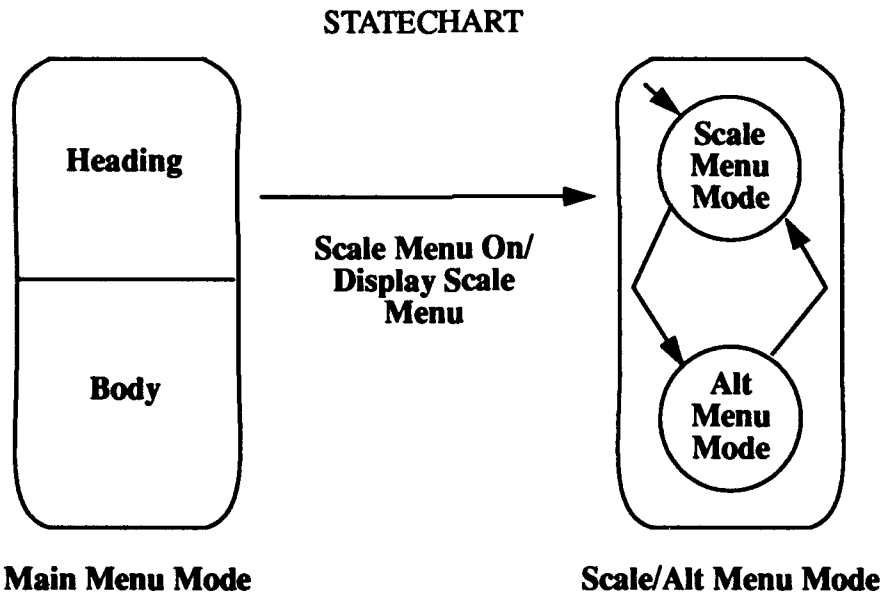


Figure 3-11. A Sample Statechart

3.6.2 Mapping Harel Statecharts to Metamodel

The mapping of the statechart to the metamodel is similar to the finite state machine. Each statechart state maps to a metamodel state. States that are "or"ed together in the statechart become instances of the higher level state with an instance arc directed from the higher level state to the substates. States that are "and"ed together become components of the superstate with a composition arc directed from the superstate to the component states. The default entry state is

shown with a "Start" attribute owned by the start state. A state is designated "active" with an "active" attribute that is either true or false. Conditions in statecharts correspond to conditions in the metamodel which are shown with the attribute-relation-constant grouping. Active state dependencies on transition arcs in statecharts become persistent stimulus arcs in the metamodel. Figure 3-12 shows the mapping of the model shown in Figure 3-11 into the metamodel.

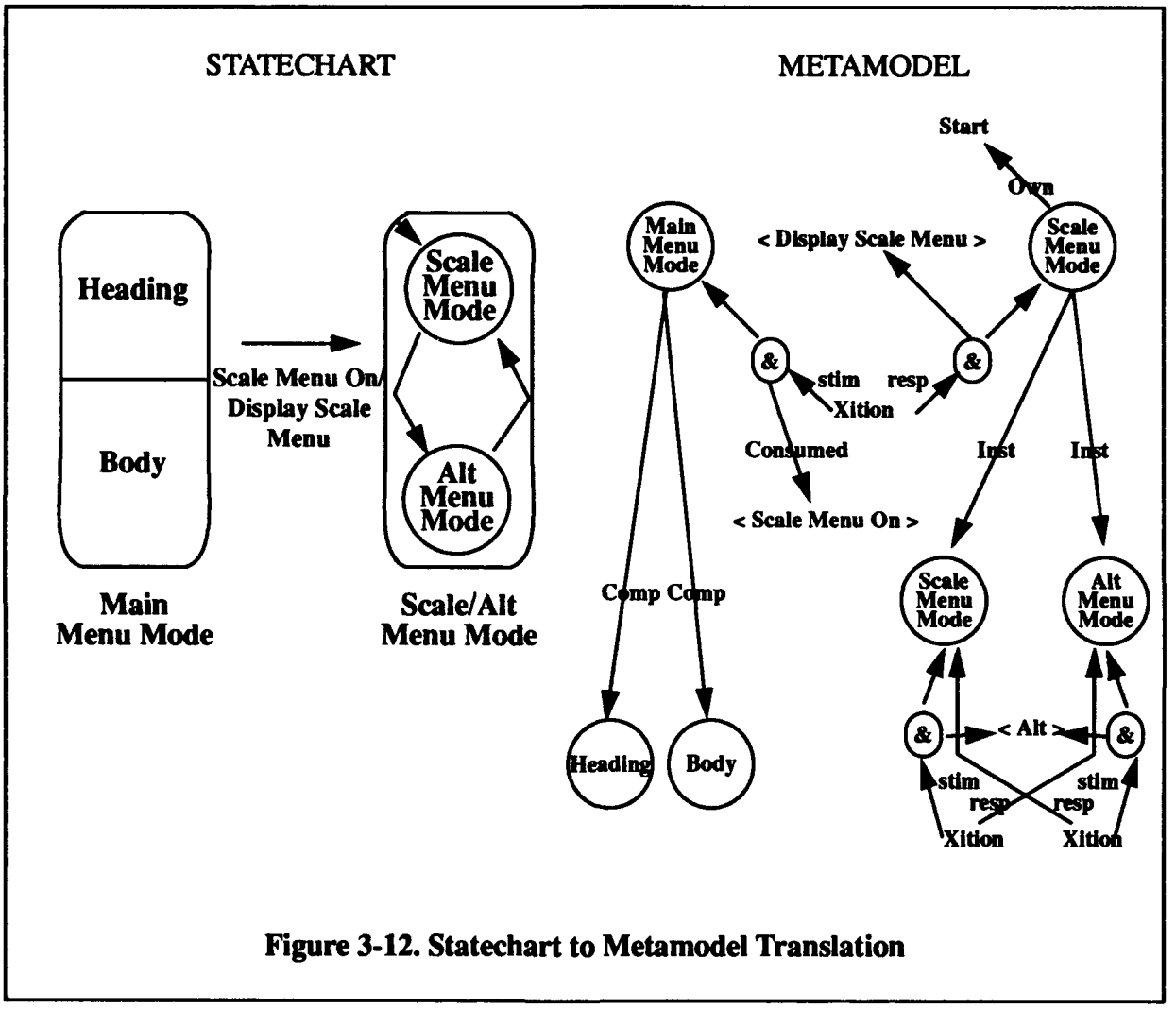


Figure 3-12. Statechart to Metamodel Translation

3.6.3 Mapping Metamodel to Harel Statechart

The mapping from the metamodel to the statechart is summarized in Table 3-6. A metamodel state becomes a statechart state. Transition nodes become FSM transition arcs. Metamodel stimuli map to a variety of statechart constructs depending on the type of node being pointed to. Metamodel responses map to either next states or response messages, depending on the type of node being pointed to.

Table 3.6 Mapping from Metamodel to Statecharts

Metamodel	Statecharts
Entity	N/A
Process	N/A
Message Between Entities and/or Processes	N/A
Attribute	N/A
State	State
Transition	Transition
Relation	Relations in Boolean Expressions
Constant	Constants in Boolean Expressions
State Ownership of Attribute "Start"	Start State
Other Ownership	N/A
Association	N/A
Composition Among States	"and" decomposition
Composition Among Other Nodes	N/A
Instantiation Among States	"or" decomposition
Instantiation Among Other Nodes	N/A
Communication	N/A
Stimulus to Message (consumed)	Stimulus
Stimulus to State (consumed)	Originating State
Stimulus to State (persistent)	Bracketed State Dependency
Stimulus to Relation (persistent)	Parenthesized Conditional Dependency
Other Stimulus	N/A
Response to State	Next State
Response to Message	Response
Other Response	N/A

3.7 Petri Nets

3.7.1 Overview

The Petri net [PET81] is a directed graph that uses two types of nodes: places and transitions. The place (represented with a circle) represents a possible condition, or state of the system. The transition (represented with a bar) represents a possible event. Directed arcs between these nodes tie together the places (conditions) and transitions (events). A token (represented with a solid circle) in a place represents the enabling of that condition. The occurrence of an event associated with a transition causes the transition to occur, if and only if all places immediately preceding the transition are enabled. Figure 3-13 shows an example of a Petri net.

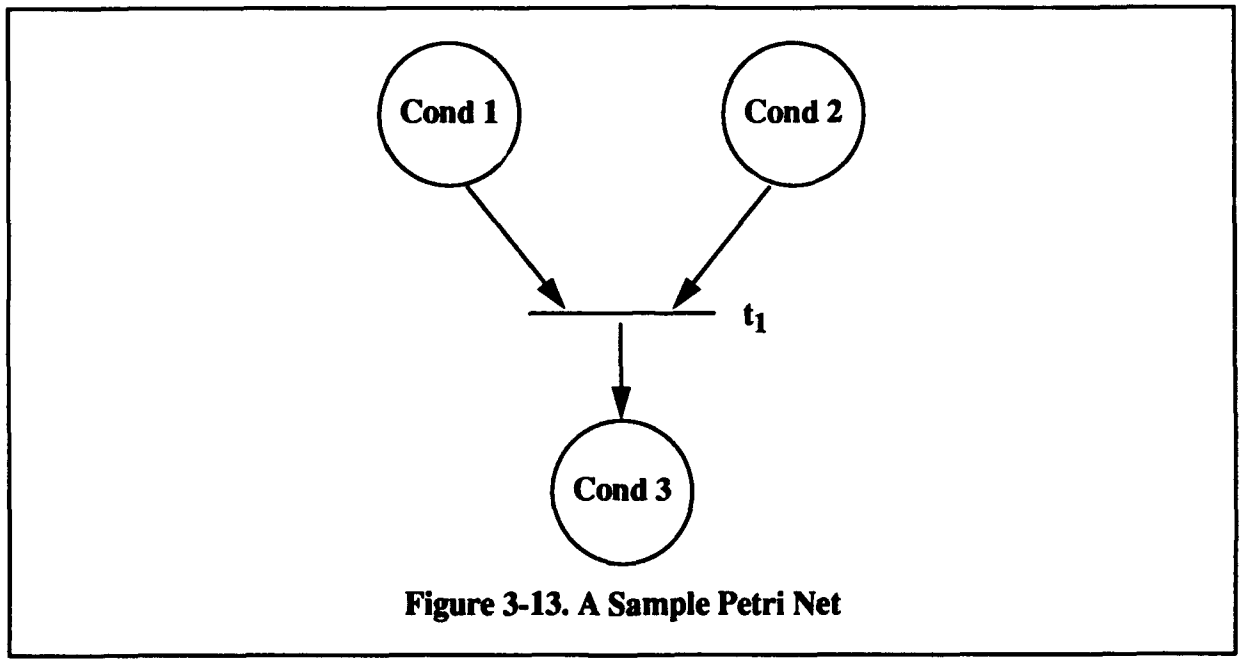


Figure 3-13. A Sample Petri Net

3.7.2 Mapping Petri Nets to Metamodel

Places become states, Petri transitions become metamodel transitions. The starting configuration becomes a collection of START attributes associated with the appropriate states. Transitions are connected via stimulus arcs to the dependent conditions and the events each represents. Transition arcs are connected via response arcs to the resultant conditions, per Figure 3-14.

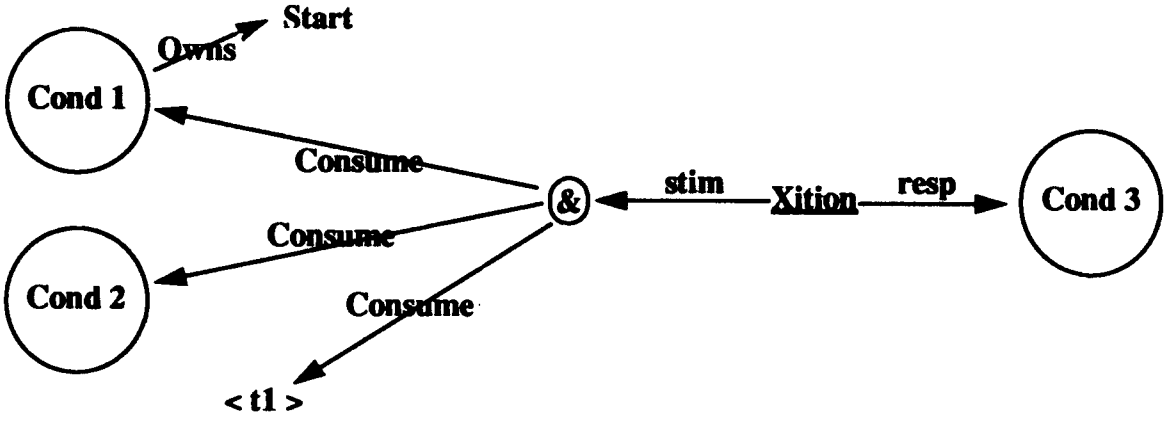


Figure 3-14. Petri Net to Metamodel Translation

3.7.3 Mapping Metamodel to Petri Nets

The mapping from the metamodel to the Petri nets is summarized in Table 3-7. Metamodel states become Petri net places. Transition nodes become Petri net transition arcs. Metamodel stimuli map to either the name of the transition or to an arc into the transition, depending on the type of node being pointed to. Metamodel responses map to arcs pointing to next states.

Table 3.7 Mapping from Metamodel to Petri Nets

Metamodel	Petri Nets
Entity	N/A
Process	N/A
Message Between Entities and/or Processes	N/A
Attribute	N/A
State	Place
Transition	Transition
Relation	N/A
Constant	N/A
State Ownership of Attribute "Start"	Initial Place Marking Member
Other Ownership	N/A
Association	N/A
Composition	N/A
Instantiation	N/A
Communication	N/A
Stimulus to Message (consumed)	Name of Transition
Stimulus to State (consumed)	Arc from Place to Transition
Other Stimulus	N/A
Response to State	Arc from Transition to Place
Other Response	N/A

3.8 Decision Trees and Tables

3.8.1 Overview

Decision trees and tables capture the causal relationships between a set of conditions and a system response. A decision tree, as shown in Figure 3-15, displays this graphically, and clearly shows how the truth or falsity of each of the conditions would invoke the appropriate action. In a decision table, as shown in Figure 3-16, rows are constructed for each condition and for each possible system action. A column is constructed for each possible combination of conditions. At the bottom of each column, the appropriate actions are indicated.

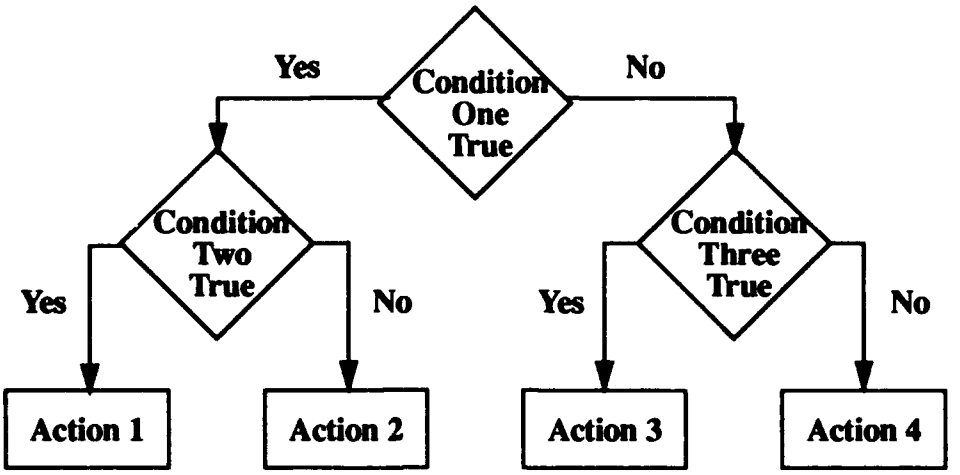


Figure 3-15. A Sample Decision Tree

Case	Rule 1	Rule 2	Rule 3	Rule 4
Condition 1	Yes	Yes	No	No
Condition 2	Yes	No		
Condition 3			Yes	No
Action 1	X			
Action 2		X		
Action 3			X	
Action 4				X

Figure 3-16. A Sample Decision Table

3.8.2 Mapping Decision Trees and Tables to Metamodel

Each of the condition rows of a decision table (i.e., cond1 to cond4 in Figure 3-17) maps into a relational expression, a message node, or a state node in the metamodel. (If the row label is already defined to be a state or message, then it remains a state or message; otherwise it becomes a relational expression.) In Figure 3-17, let us assume that cond1 through cond4 have previously been defined as states. Each of the action rows (i.e., action1 and action2 in Figure 3-17) maps into

a relational expression, a message node, or a state node in the metamodel. (If the row label is already defined to be a state or message, then it remains a state or message; otherwise it becomes a relational expression.) In Figure 3-17, let us assume that action1 and action2 have previously been defined as messages. Each column (i.e., columns 1 to 3 in Figure 3-17) maps into a transition node, whose stimulus arcs point in a persistent manner to each of the relations, messages, and states with yes's in their rows, and whose response arcs point to all the relation, message, or state nodes with x's in their rows.

DECISION TABLE

Case	1	2	3
Cond 1	Yes	Yes	...
Cond 2	Yes	Yes	...
Cond 3	Yes	Yes	...
Cond 4	Yes	No	
Action 1	X	X	
Action 2		X	

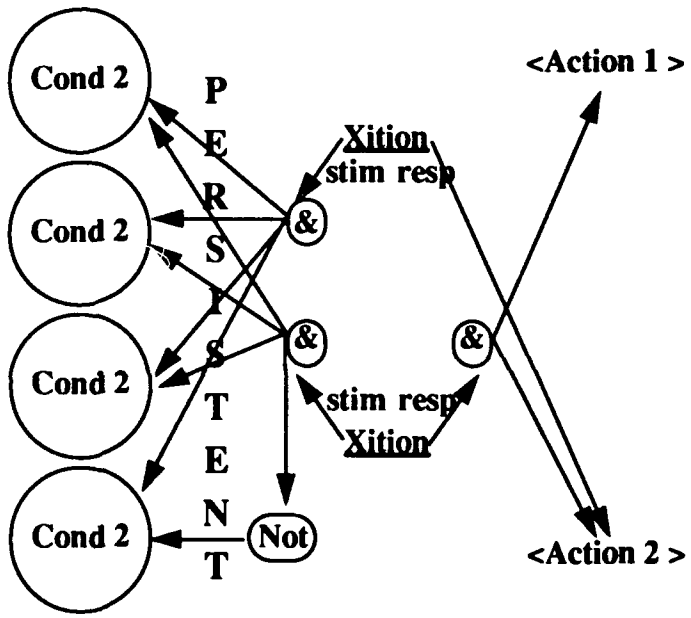


Figure 3-17. Decision Tables to Metamodel Translation Examples

3.8.3 Mapping Metamodel to Decision Trees and Tables

Transition nodes become table columns. Metamodel persistent stimuli map to condition rows in the table. Metamodel responses map to action rows in the table. See Table 3-8 which follows.

Table 3.8 Mapping from Metamodel to Decision Tables

Metamodel	Decision Tables
Entity	N/A
Process	N/A
Message Between Entities and/or Processes	N/A
Attribute	N/A
State	N/A
Transition	Column
Relation	Relation in Row Header
Constant	Constant in Row Header
Ownership	N/A
Association	N/A
Composition	N/A
Instantiation	N/A
Communication	N/A
Stimulus to Message (persistent)	Condition Row
Stimulus to State (persistent)	Condition Row
Other Stimulus	N/A
Response	Action Row

SECTION 4

4. Further Metamodel Development

The metamodel is the first step in the automated support for multiple views of requirements. The definition of the metamodel is fairly complete at this point. However, that completion will not be final until experience is gained (1) adding other requirements notations and witnessing no changes to the metamodel, and (2) applying it to an actual system. After these two validation steps, it would be appropriate to construct a multiple-view requirements engineering environment based on the metamodel.

4.1 Application to Actual System

Thus far the metamodel has only been applied to a small system. The application of the metamodel in the specification of a full system would provide an opportunity to verify the completeness of the metamodel. It would also demonstrate how the metamodel can be used to detect inconsistencies and ambiguities in a requirements specification. This experience would also address issues such as how the metamodel would handle the size and complexity of a large system, or one with real-time constraints or a distributed architecture.

4.2 Requirements Engineering Environment

To use the metamodel as intended, a prototype requirements engineering environment should be constructed that uses the refined metamodel as its basis. The objective of this environment called TINA would be to enable a requirements writer to specify requirements using a broad set of notations with the following specific advantages over more traditional requirements writing approaches [DAV91]:

- Method Independence. A system development project would be able to select those requirements notations it considers most applicable. In addition, each member of the requirements team could select the particular notation for his/her subproblem.

- View Consistency. Given a project that uses a variety of requirements notations, consistency can be determined among the views. The metamodel would provide a basis to compare and contrast different views of the same set of requirements.

- View Synthesis. Given a set of requirements in a specific notation, the requirements writer would be able to generate automatically those same requirements in other notations. The level with which the new requirements "view" can be synthesized would depend upon the degree of overlap and correspondence between the relevant notations. Table 4-1 provides some idea as to the degrees of overlap.

- Documentation Production. The documentation of requirements, especially those following a particular standard (e.g., DOD-STD-2167A's MCCR-DI-80027 [DOD88]), can be considered as being just one more view of requirements, i.e., just one more face on the Lucite box.

- Multiple System Interoperability. Suppose two or three existing systems need to be integrated, but they were specified using different techniques. Using TINA, we can integrate their specifications to find inconsistencies prior to attempting their actual interface.

- Requirements Completeness. In addition to integrating requirements models, an objective of this work is to determine what elements of the requirements domain are necessary to define requirements in a complete and consistent manner. If that is achieved, we may be able to define a "minimal cover" of requirements elements from which all requirements views could be synthesized.

Table 4.1 Overlap Between Notations

Metamodel	DFD	Ward	Hatley	OOA	FSM	Harel	P-Net	DTs
Entity	X	X	X	X				
Process	X	X	X	X				
Message	X	X	X					
Attribute				X				
State		X	X		X	X	X	
Transition		X	X		X	X	X	X
Relation						X		X
Constant						X		X
Ownership				X	X	X	X	
Association				X				
Composition	X	X	X	X		X		
Instantiation				X		X		
Communication	X	X	X	X				
Stimulus (persistent)						X		X
Stimulus (consumed)		X	X		X	X	X	
Response		X	X		X	X	X	X

REFERENCES

[ABB89] Abbott, D., "KBSA's Requirements Assistant and Aerospace Industry Needs," *Fourth Annual Knowledge Based Software Assistant Conference*, Rome Air Development Center: Griffis Air Force Base, New York, September 1989.

[ARM91] U.S. Army, "Operational Requirements for Automated Capabilities," Draft Pamphlet written by SET Committee under the leadership of Phil Casey, TRADOC, 1991.

[BLA89] Black, H., et al, *Requirements Engineering and Rapid Prototyping Workshop Proceedings*, U.S. Army Communications-Electronics Command, Fort Monmouth, New Jersey, 1989.

[COA91] Coad, P., and E. Yourdon, *Object-Oriented Analysis*, 2nd ed., Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[CSC90] Computer Sciences Corporation, *Flight Software Systems Branch (FSSB) Software Standards and Procedures (FSSP) Manual*, Standards 800, 801, 802, 803, and 804, March 1990.

[DAV79] Davis, A., and T. Rauscher, "Formal Techniques and Automatic Processing to Ensure Correctness in Requirements Specifications," *IEEE Specifications of Reliable Software Conference*, Washington, D.C.,: Computer Society Press of the Institute of Electrical and Electronics Engineers, 1979.

[DAV80] Davis, A., "Automating the Requirements Phase: Benefits to Later Phases of the Software Life Cycle," *IEEE COMPSAC '80*, Washington, D.C.: Computer Society Press of the Institute of Electrical and Electronics Engineers, 1980.

[DAV90a] Davis, A., *Software Requirements Analysis and Specification*, Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[DAV90b] Davis, A., and K. Jordan, *Requirements Engineering Metamodel: A Unified View of Requirements*, George Mason University, Fairfax, Virginia, September 1990.

[DAV91] Davis, A., and K. Jordan, "Some Ideas for a Method-Independent Requirements Environment," *24th Hawaiian International Conference of Systems Sciences*, January 1991.

[DEM79] DeMarco, T., *Structured Analysis and System Specification*, Englewood Cliffs, New Jersey, Prentice-Hall, 1979.

[DOD88] U. S. Department of Defense, Military Standard: *Defense System Software Development, DOD-STD-2167A*, Washington, D.C., February 1988.

[ELE89] Elefante, D., "An Overview of RADC's Knowledge Based Software Assistant Program," *Fourth Annual Knowledge Based Software Assistant Conference*, Rome Air Development Center: Griffis Air Force Base, New York, September 1989.

[GAN79] Gane, C., and T. Sarson, *Structured Systems Analysis: Tools and Techniques*, Englewood Cliffs, New Jersey, Prentice-Hall, 1979.

[HAR87] Harel, D., "Statecharts: A Visual Formalism for Complex Systems," *Science of Computing* (1987): 231-74.

[HAT87] Hatley, D., and I. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House Publishing, New York, New York, 1987.

[ISS90] International Software Systems, Inc., *Software Requirements Specification for the Requirements Engineering Environment Development*, Rome Air Development Center Contract No. F30602-89-C-0200, Austin, Texas, August 1990.

[JOR91] Jordan, K., and A. Davis, "Requirements Engineering Metamodel: An Integrated View of Requirements," *15th International Computer Software and Applications Conference COMPSAC '91*, Tokyo, Japan, September 1991.

[MEY91] Meyers, S., "Difficulties in Integrating Multiview Development Systems," *IEEE Software*, 8, 1 (January 1991).

[PET81] Peterson, J., *Petri Net Theory and the Modeling of Systems*, Englewood Cliffs, New Jersey: Prentice Hall, 1981.

[RUM91] Rumbaugh, J., et al., *Object-Oriented Modeling and Design*, Englewood Cliffs, New Jersey: Prentice-Hall, 1991.

[WAR85] Ward, P., and S. Mellor, *Structured Development for Real-Time Systems*, Englewood Cliffs, New Jersey: Prentice-Hall, 1985.

[YOU89] Yourdon, E., *Modern Structured Analysis*, Prentice-Hall, Englewood Cliffs, New Jersey, 1989.